Modellgetriebene Entwicklung adaptiver, komponentenbasierter Mashup-Anwendungen

DISSERTATION

zur Erlangung des akademischen Grades
Doktoringenieur (Dr.-Ing.)

vorgelegt an der
Technischen Universität Dresden
Fakultät Informatik

eingereicht von

Dipl.-Medieninf. Stefan Pietschmann
geboren am 03. Oktober 1980 in Dresden

Gutachter:

- Prof. Dr.-Ing. Klaus Meißner (Technische Universität Dresden)
- Prof. Dr.-Ing. Martin Gaedke (Technische Universität Chemnitz)

Tag der Verteidigung: 13.12.2012

TECHNISCHE UNIVERSITÄT DRESDEN

Dresden im Juli 2012

Zusammenfassung

Mit dem Wandel des Internets zu einer universellen Softwareplattform sind die Möglichkeiten und Fähigkeiten von Webanwendungen zwar rasant gestiegen. Gleichzeitig gestaltet sich ihre Entwicklung jedoch zunehmend aufwändig und komplex, was dem Wunsch nach immer kürzeren Entwicklungszyklen für möglichst situative, bedarfsgerechte Lösungen entgegensteht. Bestehende Ansätze aus Forschung und Technik, insbesondere im Umfeld der serviceorientierten Architekturen und Mashups, werden diesen Problemen bislang nicht ausreichend gerecht.

Deshalb werden in dieser Dissertation neue Konzepte für die modellgetriebene Entwicklung und Bereitstellung von Webanwendungen vorgestellt. Die zugrunde liegende Idee besteht darin, das Paradigma der Serviceorientierung auf die Präsentationsebene zu erweitern. So sollen erstmals – neben Daten- und Geschäftslogik – auch Teile der Anwendungsoberfläche in Form wiederverwendbarer Komponenten über Dienste bereitgestellt werden. Anwendungen sollen somit über alle Anwendungsebenen hinweg nach einheitlichen Prinzipien „komponiert“ werden können.

Den ersten Schwerpunkt der Arbeit bilden die entsprechenden *universellen Modellierungskonzepte* für Komponenten und Kompositionen. Sie erlauben u. a. die plattformunabhängige Beschreibung von Anwendungen als Komposition der o. g. Komponenten. Durch die Abstraktion und entsprechende Autorenwerkzeuge wird die Entwicklung so auch für Domänenexperten bzw. Nicht-Programmierer möglich.

Der zweite Schwerpunkt liegt auf dem *kontextadaptiven Integrationsprozess* von Komponenten und der zugehörigen, *serviceorientierten Referenzarchitektur*. Sie ermöglichen die dynamische Suche, Bindung und Konfiguration von Komponenten, d. h. auf Basis der o. g. Abstraktionen können genau die Anwendungskomponenten geladen und ausgeführt werden, die für den vorliegenden Nutzer-, Nutzungs- und Endgerätekontext am geeignetsten sind. Der dritte Schwerpunkt adressiert die *Kontextadaptivität* der kompositen Anwendungen in Form von Konzepten zur aspektorientierten Definition von adaptivem Verhalten im Modell und dessen Umsetzung zur Laufzeit. In Abhängigkeit von Kontextänderungen können so Rekonfigurationen von Komponenten, ihr Austausch oder Veränderungen an der Komposition, z. B. am Layout oder dem Datenfluss, automatisch durchgesetzt werden.

Alle vorgestellten Konzepte wurden durch prototypische Implementierungen praktisch untermauert. Anhand diverser Anwendungsbeispiele konnten ihre Validität und Praktikabilität – von der Modellierung im Autorenwerkzeug bis zur Ausführung und dynamischen Anpassung – nachgewiesen werden.

Die vorliegende Dissertation liefert folglich eine Antwort auf die Frage, wie zukünftige Web- bzw. Mashup-Anwendungen zeit- und kostengünstig entwickelt sowie zuverlässig und performant ausgeführt werden können. Die geschaffenen Konzepte bilden gleichermaßen die Grundlage für eine Vielzahl an Folgearbeiten.

Danksagung

Diese Dissertation ist das Resultat vieler Jahre Arbeit und sie wäre nicht möglich gewesen ohne die Unterstützung einer Vielzahl von Menschen, denen ich an dieser Stelle herzlich danken möchte.
Zunächst geht mein Dank an Prof. Dr.-Ing. Klaus Meißner für die Betreuung der Arbeit und die Chance, mit einigem Gestaltungsspielraum in diesem interessanten und bewegten Themengebiet arbeiten zu können. Sein kritisches Hinterfragen und seine Anregungen haben maßgeblich zur Reifung der Ergebnisse beigetragen. Prof. Dr.-Ing. Martin Gaedke gilt mein herzlicher Dank für die Bereitschaft, diese Arbeit zu begutachten und für seine Hinweise und hilfreichen Kommentare, insbesondere in der Strukturierungsphase.

Natürlich möchte ich mich auch bei all den Kollegen bedanken, die mich in den zurückliegenden Jahren am Lehrstuhl begleitet haben, und die mir sowohl inhaltlich als auch freundschaftlich eine Stütze waren. Der erste Dank gebührt Michael Hinz, der mich als studentische Hilfskraft, Beleg- und Diplomstudenten an das wissenschaftliche Arbeiten heranführte und in den Anfangsjahren als Mitarbeiter begleitet hat. Weiterhin danke ich all den Kollegen, die selbst aktiv zum Erfolg der Arbeit und dem angeschlossenen Forschungsprojekt beigetragen haben, oder die auf die hier vorgestellten Ergebnisse aufbauen. Namentlich möchte ich an dieser Stelle Martin Voigt hervorheben, der das Forschungsthema von Anfang an – zunächst als Student und später als Mitarbeiter – begleitet und auch inhaltlich vorangetrieben hat. Er fand stets die Zeit für konstruktive Kritik und Gespräche. Gleichermaßen gilt mein Dank Annett Mitschick für die zahlreichen wissenschaftlichen und nicht-wissenschaftlichen Diskussionen und Hinweise. Beide haben diese Arbeit unter viel Zeitaufwand Korrektur gelesen und mich auf die Schwachpunkte hingewiesen - ihr wart mir eine große Hilfe.
Neben der fachlichen Unterstützung weiß ich auch das kollegiale, freundschaftliche Umfeld am Lehrstuhl zu schätzen. In diesem Zusammenhang möchte ich speziell Ramona Behling und Udo Wähner für ihre „Rückendeckung“ auf organisatorischer und auf menschlicher Ebene danken.

Nicht zuletzt gebührt mein großer Dank all den Studenten, die sich mit mir den wissenschaftlichen Herausforderungen gewidmet haben und die – sei es durch Beleg- und Diplomarbeiten, in Seminaren oder Komplexpraktika – tatkräftig zu den vorgestellten Konzepten beitragen konnten. Stellvertretend möchte ich mich bei Carsten Radeck und Johannes Waltsgott, die mich später als Kollegen begleitet haben, sowie bei Jan Reimann und Robert Wende bedanken.

Mein abschließender Dank gilt meinen Eltern, auf deren fortwährende Unterstützung in jeglicher Beziehung ich mich über die Jahre des Studiums und der Promotion stets verlassen konnte.

Inhaltsverzeichnis

Abbildungsverzeichnis

Verzeichnis der Codebeispiele

Abkürzungsverzeichnis

AMACONT . SystemArchitektur für Multimedialen Adaptiven WebCONTent
API . . . . . Application Programming Interface

BPEL . . . . Business Process Execution Language

CASE . . . . Computer-Aided Software Engineering
CBSE . . . . Component Based Software Engineering
CoRe . . . . Component Repository
CRM . . . . Customer Relationship Management
CroCo . . . Cross-Application Context Management
CRUD . . . Create Read Update Delete
CRUISe . . . Composition of Rich User Interface Services
CSA . . . . . Context Service Adapter
CSS . . . . . Cascading Style Sheets
CTT . . . . . ConcurTaskTree

DCCI . . . . Delivery Context: Client Interfaces
DEMISA . . Development Methods for process-drIven composite maShup Applications
DOM . . . . Document Object Model
DSL . . . . . Domain Specific Language

EAI . . . . . Enterprise Application Integration
ECA . . . . . Event Condition Action
EDYRA . . . Engineering of Do-it-Yourself Rich Internet Applications
EII . . . . . Enterprise Information Integration
EMF . . . . Eclipse Modeling Framework
EMML . . . Enterprise Mashup Markup Language
EUD . . . . End User Development

FIFO . . . . First In First Out

GRDDL . . . Gleaning Resource Descriptions from Dialects of Languages
GWT . . . . Google Web Toolkit

HTML . . . . HyperText Markup Language
HTTP . . . . HyperText Transfer Protocol

JAR . . . . . Java ARrchive
JAXB . . . . Java Architecture for XML Binding

JSON . . . . JavaScript Object Notation
JSP . . . . . JavaServer Pages

K-IMM . . . Knowledge through Intelligent Media Management

LBS . . . . . Location Based Services
LGPL . . . . GNU Lesser General Public License

MCDL . . . Mashup Component Description Language
MCDO . . . Mashup Component Description Ontology
MDA . . . . Model Driven Architecture
MDD . . . . Model Driven Development
MDL . . . . mashArt Description Language
MOF . . . . Meta Object Facility
MRE . . . . Mashup Runtime Environment
MVC . . . . Model View Controller

NFP . . . . . Non Functional Properties
NIST . . . . National Institute of Standards and Technology

OASIS . . . Organization for the Advancement of Structured Information Standards
OCL . . . . . Object Constraint Language
OSGi . . . . Open Services Gateway initiative
OSL . . . . . Open Software License
OWL . . . . Web Ontology Language

PaaS . . . . Platform as a Service
PIM . . . . . Platform Independent Model
PML . . . . . Presentation Modeling Language

QoS . . . . . Quality of Service
QVT . . . . . Query View Transformation

RAP . . . . . Rich Ajax Platform
RDF . . . . . Resource Description Framework
RDFa . . . . RDF in attributes
REST . . . . Representational State Transfer
RIA . . . . . Rich Internet Application
RSS . . . . . Really Simple Syndication
RWT . . . . RAP Widget Toolkit

SaaS . . . . Software as a Service
SAWSDL . . Semantic Annotations for WSDL and XML
SCA . . . . . Service Component Architecture
SMCDL . . . Semantic MCDL
SOA . . . . . Serviceorientierte Architektur
SOAP . . . . Simple Object Access Protocol
SOC . . . . . Service Oriented Computing

SoC . . . . . Separation of Concerns
SOP . . . . . Same Origin Policy
SOSE . . . . Service Oriented Software Engineering
SPARQL . . SPARQL Query Language for RDF
SWRL . . . . Semantic Web Rule Language
SWS . . . . Semantic Web Services
SWT . . . . Standard Widget Toolkit

TLC . . . . . Task List Client
TSR . . . . . Thin Server Runtime
TyRe . . . . Semantic Type Repository

UCL . . . . . Universal Composition Language
UI . . . . . User Interface
UIML . . . . User Interface Modeling Language
UIS . . . . . User Interface Service
UISDL . . . User Interface Service Description Language
UML . . . . Unified Modeling Language
URI . . . . . Uniform Resource Identifier
URL . . . . . Uniform Resource Locator
UWE . . . . UML-based Web Engineering

VCS . . . . . Virtual Consulting Services
VVO . . . . . Verkehrsverbund Oberelbe

W3C . . . . World Wide Web Consortium
WADL . . . . Web Application Description Language
WAR . . . . Web Application Ressource
WCML . . . WebComposition Markup Language
WSDL . . . Web Service Description Language
WSML . . . Web Service Modeling Language
WSMO . . . Web Service Modeling Ontology
WURFL . . . Wireless Universal Resource FiLe
WWW . . . . World Wide Web

XAML . . . . eXtensible Application Markup Language
XHR . . . . . XMLHttpRequest
XHTML . . . eXtensible HyperText Markup Language
XMI . . . . . XML Metadata Interchange
XML . . . . . eXtensible Markup Language
XSLT . . . . eXtensible Stylesheet Language Transformation

1 Einleitung

Das Internet ist im Wandel. Diente es vor zehn Jahren noch vorrangig der Publikation und Verwaltung von Daten in Form von Hypertext-basierten Dokumenten im World Wide Web (WWW), so offenbart es inzwischen zunehmend den Charakter einer universellen Anwendungsplattform. Eine Vielzahl konventioneller Applikationen wird heute in Form von Software as a Service (SaaS), d. h. als Dienstleistung über das Web angeboten. Dies ist – neben offensichtlichen Vorteilen, wie dem zeit-, plattform- und ortsunabhängigen Zugriff – auch mit der Etablierung neuer Geschäftsmodelle wie *pay per use* verbunden.

Parallel zu diesen technischen Trends ist auch die IT-Welt im Wandel: In der Beziehung zwischen IT- und Fachabteilungen bzw. zwischen Software-Entwicklern und -Nutzern zeichnet sich eine Änderung der Rollenverteilung ab. Entwickler stellen zunehmend Plattformen bereit, die *Power User* und Domänenexperten in die Lage versetzen, konkrete Aufgaben und Probleme selbst auf Software-Lösungen abzubilden und Endanwendern zur Verfügung zu stellen. Teilweise wird der Endnutzer selbst zum Entwickler – man spricht vom End User Development (EUD).

Grundlage hierfür bietet das sog. *Programmable Web* [Maximilien et al., 2007] bzw. das Internet der Dienste und Dienstleistungen. Gemäß der Prinzipien der Separation of Concerns (SoC) ermöglicht es die entkoppelte Entwicklung und Bereitstellung von Daten, Geschäftslogik und ganzen Anwendungen, die über generische Schnittstellen technologieunabhängig und beliebig oft wiederverwendet und zu komplexeren Anwendungen komponiert werden können.

Das Modell der *Mashups* greift diesen Ansatz auf. Ihr Ziel ist die „leichtgewichtige" Kombination eben jener verteilten Web-Ressourcen, d. h. von Inhalten und Funktionalitäten lokaler und entfernter Quellen, zur Lösung konkreter Anwendungsprobleme. Im Gegensatz zur traditionellen Softwareentwicklung ist der Mashup-Lebenszyklus stark verkürzt, da der Fokus auf der möglichst zeit- und kostengünstigen Erstellung situativer Anwendungen – u. U. auch durch Nicht-Programmierer – aus vorgefertigten Bausteinen liegt. Die immense Bedeutung allein für das geschäftliche Umfeld lässt sich anhand des geschätzten Marktvolumens zwischen 682 Millionen (Forrester [Young et al., 2008]) und 1,74 Milliarden $ (Business Insights [Absalom, 2009]) weltweit für das Jahr 2013 erahnen.

Zwischen den Idealen von Mashups und der Erstellung heutiger SaaS-Lösungen liegen jedoch Welten: Während für die Entwicklung sowohl konventioneller Desktop-Software als auch klassischer Hypermedia-Anwendungen etablierte Vorgehensmodelle existieren, steckt die strukturierte Entwicklung interaktiver, *kompositer* Webanwendungen in den Kinderschuhen. Die zugrunde liegenden Konzepte rund um Serviceorientierte Architekturen (SOAs) haben zwar die Integration auf Daten- und Anwendungsebene mit Hilfe von Standardisierungsbemühungen und Frameworks vereinfacht, bieten jedoch keine Konzepte für die Präsentationsebene, d. h. zur Nutzerinteraktion mit Dienstkompositionen.

Für die Interaktion mit Diensten existieren verschiedene Herangehensweisen [Steger und Kappert, 2008]: „Dienste ohne Gesicht“ können nur über die von Ihnen gebotenen Schnittstellen genutzt werden, was ein technisches Verständnis sowie entsprechende Werkzeuge zum Umgang mit Protokollen und Formaten wie Simple Object Access Protocol (SOAP) und JavaScript Object Notation (JSON) nötig macht. Dieser Ansatz ist für Nicht-Programmierer folglich ungeeignet. „Dienste mit kontextfreiem Gesicht“ stellen für die Interaktion vorgefertigte HTML-Fragmente, *Gadgets* oder *Portlets* zur Verfügung, die fest mit den Dienstaufrufen gekoppelt sind. Neben der Frage, wer diese entwickelt, bleibt das Problem, dass nur mit einem Dienst interagiert werden kann, und die Oberflächen statisch, ohne jeglichen Bezug zur Anwendung und zum Anwendungskontext bleiben. Die häufigste Form der Interaktion erfolgt „mit kontextbezogenen Gesicht“, d. h. es werden dedizierte Benutzerschnittstellen bzw. Rich Internet Applications (RIAs) entwickelt, die mehrere Dienste anbinden und miteinander koppeln. Hier schließt sich der Kreis zu den genannten SaaS-Lösungen.

Beim Design derartiger interaktiver Webanwendungen sehen sich Entwickler vielen Herausforderungen gegenüber, die dem dezentralen und mobilen Zugriff geschuldet sind. Hauptsächliches Problem stellen die heterogenen Nutzer-, Nutzungs- und Endgerätekontexte dar, an die sich Anwendungen zwingend anpassen müssen, um ihre Funktionalität und nicht zuletzt eine gute Usability zu gewährleisten. Dieser Anspruch hat den Entwicklungsprozess insbesondere an der Schnittstelle zwischen Nutzer und Anwendung, d. h. auf der Präsentationsebene, dramatisch verkompliziert. Hinzu kommt, dass die User Interface (UI)-Entwicklung in den letzten Jahren technologisch an Komplexität gewonnen hat. Unzählige verschiedene Plattformen und Frameworks führen bei der Implementierung zwangsläufig dazu, dass es den geschaffenen Oberflächen und ihren Bestandteilen an Interoperabilität, Wiederverwendbarkeit und technologischer Zukunftssicherheit mangelt.

Die fehlende Anwendung der SOA-Prinzipien auf dieser Ebene führt letztlich zu „throw-away and recreate“ UIs für spezifische Anwendungskontexte [@Davies, 2006]. RIAs bieten zwar den besten Zugang zu Diensten und Dienstkompositionen, sie sind jedoch komplex, unflexibel und mit dem zeitlich und finanziell größten Entwicklungsaufwand verbunden. Man geht davon aus, dass der Anteil der UI-Entwicklung und -Wartung am Gesamtaufwand der Softwareentwicklung bei 50% [Myers und Rosson, 1992] bis 70% [Kleshchev und Gribovy, 2003] liegt. Diese Zahlen variieren selbstverständlich in Abhängigkeit von Aspekten wie der Anwendungskomplexität, der Heterogenität der Zielgruppe oder der genutzten Technologie. In jedem Fall zeigen sie den tendenziell hohen Aufwand bei der Umsetzung von Benutzerschnittstellen – gerade in Anbetracht der Vereinfachungen im Backend, z. B. durch den Einsatz von Web Services.

Vor diesem Hintergrund ist im Mashup-Umfeld ein Trend hin zur sog. *Presentation Integration* zu erkennen. Dahinter verbirgt sich die Idee, die Dienst- bzw. Anwendungskomposition auf der Präsentationsebene zu unterstützen. Kompositionsobjekte sind in diesem Fall nicht mehr Web Services, sondern Web-Anwendungen, Portlets, Widgets, o.ä. [Ngu et al., 2010]. Diese werden i. d. R. grafisch miteinander verknüpft und somit eine Anwendungsintegration erreicht. Ein derartiges Kompositionsparadigma ermöglicht neuartige Entwicklungsprozesse, an denen auch der Nutzer selbst beteiligt werden kann, und ist gerade für Unternehmen zunehmend von Interesse [Jhingran, 2006]. Die Konzepte der klassischen SOA sind hierfür freilich nicht ausreichend, da sie keinerlei interaktive Komponenten vorsehen.
Im Hinblick auf die skizzierten Probleme und Herausforderungen der UI-Entwicklung hat die *Presentation Integration* einen entscheidenden Nachteil: Die Integration ganzer „Anwendungsblöcke" widerspricht dem Prinzip der *Trennung der Verantwortlichkeiten*. Die feste Kopplung von Daten und Geschäftslogik mit der Benutzerschnittstelle führt zu einem Verlust an Flexibilität und macht die Homogenisierung und die Kontextadaptivität insbesondere der UI weitgehend unmöglich.

In Anbetracht der aufgeworfenen Probleme stellen sich grundsätzliche Fragen: Wie können moderne, interaktive Anwendungen (i. S. v. Interaktion mit dem Nutzer) aus Komponenten bzw. Diensten zusammengesetzt werden? Welche Möglichkeiten der plattformunabhängigen Entwicklung und Modellierung gibt es? Welche Systeminfrastruktur ist für derartige Kompositionen nach dem Vorbild der SOA nötig? Und welche Konzepte müssen auf der Modellierungs- und Infrastrukturebene entwickelt werden, um Kontextadaptivität zu unterstützen? All diese Fragen sind bislang nur unzureichend beantwortet worden. Die folgenden Abschnitte widmen sich deshalb der Identifikation der Kernprobleme, der Formulierung der Thesen sowie der Vorstellung der Forschungsziele für die vorliegende Arbeit.

1.1 Problemdefinition, Thesen und Forschungsziele

Aus der kurzen Motivation und Darstellung des Anwendungsfeldes ist bereits ersichtlich, dass der Einsatz kompositer Webanwendungen für Autoren und Nutzer einige Vorteile verspricht. Ihre Entwicklung und Bereitstellung werfen aber eine Reihe komplexer Forschungsfragen auf, die durch bestehende Ansätze bislang nicht ausreichend beantwortet werden.
Die folgenden Abschnitte legen dar, welchen konkreten Problemstellungen sich diese Dissertation widmet, welche Thesen der Arbeit zugrunde liegen und welche wissenschaftlichen Ziele vor diesem Hintergrund verfolgt werden.

1.1.1 Probleme

Die grundlegenden Probleme, die diese Dissertation adressiert, sind die folgenden:

- Serviceorientierte Web- bzw. Mashup-Anwendungen werden i. d. R. programmatisch entwickelt, da die Dienstprinzipien zwar die Verteilung und einfache Kopplung im Backend unterstützen, auf der Präsentationsebene allerdings keine Anwendung finden. Die Erstellung und Wartung der Oberflächen und

ihre Verknüpfung mit Daten und Geschäftslogik im Backend stellt sich als äußerst zeit- und kostenintensiv dar.

- Die Entwicklung derartiger Anwendungen ist insbesondere auf der Präsentationsebene an konkrete Technologien und Herstellerplattformen gebunden. Dies geht zu Lasten der Interoperabilität und Wiederverwendbarkeit, d. h. Lösungen müssen plattform- und anwendungsspezifischen neu erstellt werden, was die o. g. Aufwände weiter steigert.
- In Verbindung mit der zunehmenden Heterogenität der Nutzer- und Nutzungskontexte ergibt sich die sog. *Impossible Equation* [Bezivin, 2011]: Eine langsam wachsende Zahl an Entwicklern sieht sich einem dramatisch steigenden Bedarf individueller Anwendungslösungen gegenüber, der mit den bestehenden Lösungsansätzen zur Abstraktion der technologischen Komplexität, zur Komponentisierung und Wiederverwendung nicht gedeckt werden kann:
  - Konzepte und Standards aus dem SOA-Umfeld beschränken sich auf prozessorientierte, automatisierte Service-Kompositionen ohne Einbeziehung von Nutzerinteraktionen bzw. -oberflächen.
  - Ansätze des Web-Engineerings beruhen auf einer dokumentenzentrierten Sichtweise, d. h. sie dienen der Modellierung von Daten und ihrer Präsentation, nicht der Komposition unabhängiger, wiederverwendbarer Anwendungsbestandteile bzw. Komponenten.
  - Mashup-Konzepte sehen ebenfalls die programmatische Entwicklung oder die Nutzung proprietärer Werkzeuge vor. Im letzteren Fall wird vorrangig die Komposition auf der Präsentationsebene unterstützt (*Presentation Integration*). Die fehlende Trennung zwischen Daten, Anwendungslogik und UI steht der Flexibilisierung der Lösungen entgegen und wirkt sich langfristig negativ auf ihre Wartbarkeit und Qualität aus.

  Der Aufwand bei der Entwicklung interaktiver Mashups und serviceorientierter Anwendungen ist somit ähnlich hoch, wie bei konventionellen Webanwendungen. Dies widerspricht dem angestrebten, möglichst schnellen und leichtgewichtigen Entwicklungsprozess.
- Die Spezifika der Präsentationsschicht bzw. interaktiver Bestandteile werden in keinem der bestehenden Kompositionsansätze ausreichend adressiert. In der Mehrzahl der Systeme wird von zustandslosen Anwendungsbestandteilen ausgegangen, deren Kopplung über die unidirektionale Verknüpfung ihrer Ein- und Ausgänge erfolgt. Weder der Zustand der Oberfläche, z. B. die Selektion bestimmter Inhalte, noch erweiterte Koordinations- und Interaktionskonzepte, z. B. zur Synchronisierung, Datenabfrage oder Nutzung von Drag-and-Drop, können mit diesen Modellen ausgedrückt werden.
- Komposite Web- bzw. Mashup-Anwendungen werden bislang als statische Verknüpfung spezifischer, verteilter Ressourcen entwickelt. Die Möglichkeiten des Service Oriented Computing (SOC), wie die Verteilung, die späte und kontextabhängige Bindung, finden dabei – insbesondere auf die UIs – keine Anwendung, da es an Konzepten für die dynamische, kontextsensitive Suche und Integration der Bestandteile zur Laufzeit fehlt. Dies schränkt die Möglichkeiten der Kontextualisierung (z. B. Auswahl kontextspezifischer Oberflächen) und

Fehlertoleranz (z. B. Austausch von Komponenten im Fehlerfall) auf vordefinierte Alternativen und die programmatische Behandlung ein.

- Webanwendungen sehen sich einer dramatischen Zunahme und Vielfalt internetfähiger, mobiler Endgeräte, sowie vielfältigen Präferenzen und Eigenschaften ihrer Nutzer gegenüber. Deshalb stellt ihre Adaptivität auch zur Laufzeit eine zunehmende Notwendigkeit dar, die allerdings sowohl den Autorenprozess als auch die Systemarchitekturen verkompliziert. Dennoch existieren keine Konzepte zur Unterstützung von Adaptivität in interaktiven Mashups.
  - Adaptionsansätze aus dem SOA-Umfeld bieten keinerlei Konzepte zur Anpassung auf der Präsentationsebene. Sie gehen von zustandslosen Diensten aus und beschränken sich auf deren Austausch und die Manipulation der vermittelten Nachrichten bzw. Daten.
  - Lösungen im Bereich *Adaptive Hypermedia* legen eine dokumentenzentrierte Sichtweise und typische Hypertext-Konzepte (Links, Seiten, etc.) zugrunde. Die Anwendbarkeit ihrer Adaptionsmethoden und -techniken (z. B. Link-Annotation) auf funktionale Kompositionen unter Berücksichtigung des *Black-Box*-Prinzips ist deshalb nicht gegeben.
  - Kompositionsansätze für interaktive Mashups lassen jegliche Unterstützung für Adaption vermissen und sind aufgrund ihrer Charakteristika auf die Konzepte aus dem SOA-Umfeld beschränkt.

  Es mangelt Mashup-Technologien somit sowohl an Konzepten, um adaptives Verhalten in abstrakter, wiederverwendbarer Form zu beschreiben, als auch an entsprechend generischen Adaptionsarchitekturen.

1.1.2 Thesen

Ausgehend von den identifizierten Problemen und Defiziten bestehender Ansätze lassen sich für die Dissertation die folgenden Thesen ableiten, die als Basis für die durch diese Dissertation zu lösenden Forschungsfragen dienen.

These 1: Durch die einheitliche und unabhängige Repräsentation von Anwendungsbestandteilen der Daten-, Geschäftslogik- und Präsentationsebene auf der Basis eines **universellen Komponentenmodells** können die von SOA bekannten Vorteile auf komposite Webanwendungen übertragen werden:

1A. Die Entwicklung und Wartung all dieser Bestandteile kann örtlich und zeitlich getrennt geschehen, was u. a. zur Senkung der Entwicklungskosten kompositer Anwendungen beiträgt und aufgrund der schnelleren, einfacheren Wartung einen Beitrag zur Qualitätssteigerung leistet.

1B. Die abstrakte, technologie- und plattformunabhängige Beschreibung von Komponenten erlaubt ihre Einbindung in verschiedene Plattformen und in unterschiedlichen Anwendungskontexten. Durch diese Wiederverwendung sinken Entwicklungsaufwand und -zeit der Anwendungen.

1C. Die Unabhängigkeit und lose Kopplung von Komponenten verschiedener Anwendungsebenen ermöglicht deren flexible, bedarfsgerechte Kombination, um beispielsweise verschiedene Oberflächen kontextabhängig für das gleiche Backend zu nutzen.

These 2: Die **universelle Komposition** auf Basis des neuartigen Komponentenmodells führt zur Vereinfachung des gesamten Entwicklungsprozesses interaktiver Mashup-Anwendungen:

2A. Der Aufwand der Anwendungserstellung wird durch die Nutzung generischer, wiederverwendbarer Komponenten reduziert. Der Entwicklungsprozess verkürzt sich im Idealfall auf deren Suche und Verknüpfung und ermöglicht die Bedienung des *Long Tail* [Anderson, 2006] in Form spezifischer, situativer Softwarelösungen.

2B. Die Etablierung eines modellgetriebenen Entwicklungsprozesses ermöglicht die plattformunabhängige Beschreibung kompositer Anwendungen. Die strukturierte Entwicklung verspricht zudem langfristig qualitativ höherwertige Lösungen als die konventionelle, programmatische Umsetzung sowie eine gesteigerte Interoperabilität und Wiederverwendbarkeit.

2C. Die Abstraktion von technologischen und Plattformeigenheiten im Anwendungsmodell erlaubt zudem Einbeziehung von Nicht-Programmierern bzw. Domänenexperten in den Erstellungsprozess.

These 3: Durch die **späte Bindung von Komponenten** zur Laufzeit lassen sich Anwendungen – einschließlich ihrer Benutzerschnittstelle – adaptiv gestalten, ohne den Entwicklungsprozess zu erschweren:

3A. Durch die Abstraktionen im Anwendungsmodell kann die Auswahl und Bindung konkreter Implementierungen von Komponenten zur Kompositions- bzw. Laufzeit erfolgen, was die Einbeziehung von Kontextinformationen in die Entscheidungsfindung ermöglicht.

3B. Die Abstraktionen führen ferner zur Vereinfachung der Entwicklung und zu einer erhöhten Flexibilität der Lösungen. Mit der Einbeziehung semantischer Technologien nach dem Vorbild der SWS können die geschilderten Interoperabilitätsprobleme zwischen syntaktisch inkompatiblen Komponenten auch auf der Präsentationsebene adressiert werden.

These 4: Adaptives Verhalten kompositer Webanwendungen lässt sich abstrahieren und als unabhängiger Teil des angestrebten Anwendungsmodells beschreiben. Die **Formalisierung von Adaptionsbelangen** und die Übertragung von Adaptionskonzepten der SOA und des *Adaptive Hypermedia* auf das universelle Komponentenmodell verspricht die folgenden Vorteile:

4A. Die abstrakte, modellbasierte Formulierung von adaptivem Verhalten stellt gegenüber der bisherigen, programmatischen Deklaration eine deutliche Vereinfachung dar. Zudem kann es anwendungs- und plattformübergreifend wiederverwendet werden.

4B. Durch den Einsatz generischer Adaptionstechniken in Verbindung mit der universellen Komposition wird es erstmals möglich, Webanwendungen auf allen Anwendungsebenen zur Laufzeit an veränderliche Kontextbedingungen anzupassen.

1.1.3 Forschungsziele

Zur Lösung der genannten Probleme und Erreichung der mit den Thesen verbundenen Vorteile, soll in dieser Arbeit die modellbasierte Entwicklung adaptiver, komponentenbasierter Webanwendungen mit besonderem Fokus auf die Anforderungen von interaktiven Mashups ermöglicht werden. Die Dissertation widmet sich deshalb drei grundlegenden, wissenschaftlichen **Herausforderungen**:

❶ **Modellierung** Die plattformunabhängige, *universelle Modellierung* von Komponenten und deren Komposition zu interaktiven Mashup-Anwendungen.

❷ **Ausführung** Die Schaffung einer modularen *Kompositionsinfrastruktur* zur dynamischen Komposition und Ausführung der modellierten Anwendungen.

❸ **Adaption** Die Unterstützung von *Kontextadaptivität* sowohl bei der Komponentenauswahl und -komposition als auch bei der Ausführung kompositer Webanwendungen.

Vor dem Hintergrund dieser Schwerpunkte sollen in der vorliegenden Arbeit die folgenden Forschungsziele erreicht werden:

Herausforderung ❶: Modellierung

Ein Ziel der Arbeit ist es, die Vorteile der Serviceorientierung durchgängig für die Erstellung interaktiver, kompositer Webanwendungen nutzbar zu machen. Dazu gilt es zunächst, ihre Bestandteile universell und generisch zu repräsentieren. Auf dieser Basis soll ihre Veröffentlichung, dynamische Anbindung und lose Kopplung nach dem Vorbild der Web Services erfolgen. Durch Abstraktionsmechanismen sollen Entwickler zudem von den Eigenheiten und der Komplexität spezifischer Plattformen abgeschirmt werden.

- ➦ **Universelles Komponentenmodell**: Es soll ein offenes, universelles Komponentenmodell für komposite Webanwendungen entwickelt werden, welches die einheitliche Repräsentation von Anwendungsbestandteilen der Daten-, Geschäftslogik- und Präsentationsebene erlaubt. Die entwickelten Abstraktionen und Kommunikationsparadigmen sollen die freie Komposition auf allen Ebenen einer Webanwendung ermöglichen. Sie sollen durch eine technologieunabhängige Beschreibung in Anlehnung an Web Service Description Language (WSDL) [@Chinnici et al., 2007] formalisiert werden, die ihre Verwaltung, Auffindung, Einbindung und Konfiguration ermöglicht.
- ➦ **Plattformunabhängiges Kompositionsmodell**: Zur plattformunabhängigen Spezifikation interaktiver Webanwendungen als Komposition der o. g. Komponenten soll ein neuartiges, belangorientiertes Metamodell entworfen werden. Dies schließt die Entwicklung von Modellkonstrukten ein, die die Integration und Konfiguration von Komponenten sowie den Daten- und Kontrollfluss der Anwendung repräsentieren. Darüber hinaus müssen spezifische Aspekte der Präsentationsebene, z. B. hinsichtlich des Layouts oder erhöhter Anforderungen an die Koordination, modellierbar sein.

Herausforderung ❷: Ausführung

Neben dem Entwicklungsprozess und den zugrunde liegenden Modellen wirft die Architektur kompositer Webanwendungen bislang unbeantwortete Forschungsfragen

auf. Hier mangelt es an einer flexiblen, offenen Infrastruktur zur Unterstützung der Komposition und Ausführung. Im Gegensatz zu proprietären Ansätzen gilt es, die gewünschte Abstraktion der Modelle auf verschiedene technologische Plattformen abzubilden – ähnlich, wie es die Nutzung von Web Services in SOA erlaubt.

- **Dynamische, kontextadaptive Bindung**: Es gilt, die SOA-Prinzipien – insbesondere hinsichtlich der losen Kopplung und dynamischen Bindung von Komponenten – auf universelle Mashup-Kompositionen zu übertragen. Dies impliziert die Konzeption eines Integrationsprozesses, welcher, ausgehend vom entwickelten Anwendungsmodell, Komponenten der verschiedenen Anwendungsebenen kontextabhängig sucht und auswählt. In diesem Zusammenhang müssen grundlegende Konzepte zur Verwaltung von Komponenten und von Kontextinformationen entwickelt werden.
- **Referenzarchitektur zur universellen Komposition**: Für die modellgerechte Komposition und Ausführung der Anwendungen soll eine flexible Kompositionsplattform konzipiert werden, die u. a. die dynamische Integration der o. g. Komponenten und ihre lose Kopplung zur Laufzeit erlaubt. Als Grundlage bedarf es einer modularen Kompositionsinfrastruktur, die über generische Schnittstellen die grundlegenden, benötigten Dienstleistungen, wie den o. g. Integrationsprozess, verschiedenen Realisierungen der Referenzplattform zur Verfügung stellt.

Herausforderung ❹: Adaption

Eine orthogonale Herausforderung, welche bislang nur unzureichend adressiert wurde, ist die Entwicklung entsprechender Adaptionskonzepte. Die damit verbundenen Fragestellungen müssen sowohl auf der Modellebene als auch bezüglich der Systemarchitektur und Laufzeitumgebung beantwortet werden.

- **Formalisierung entsprechender Adaptionstechniken**: Bestehende Adaptionsmethoden und -techniken für serviceorientierte und Hypermedia-Anwendungen sollen vor dem Hintergrund universeller Mashup-Kompositionen neu bewertet und formalisiert werden. Dies umfasst die Entwicklung geeigneter Spezifikationsmittel auf der Modellebene, um komponentenspezifische und komponentenübergreifende Adaptionsmuster möglichst abstrakt und plattformunabhängig repräsentieren zu können.
- **Kontextualisierung und dynamische Anpassung**: Schließlich soll zur Interpretation und Umsetzung des abstrakten, formalisierten Adaptionswissen die entsprechende Systemunterstützung geschaffen werden. Dies schließt die An- bzw. Einbindung entsprechender Kontexterfassungs- und verwaltungsmechanismen sowie die Realisierung der identifizierten Adaptionstechniken im Zusammenspiel mit der o. g. Laufzeitumgebung ein.

Eine Konkretisierung der Ziele in Form funktionaler und nicht-funktionaler Anforderungen erfolgt in Abschnitt 2.3. Die entwickelten Konzepte zur modellgetriebenen Entwicklung, zur Komposition und Ausführung adaptiver Mashup-Anwendungen sollen anschließend anhand ausgewählter Anwendungsszenarien erprobt und validiert werden. Grundlage hierfür bietet die Schaffung einer integrierten Gesamtlösung, die wesentliche Teile der Konzepte prototypisch umsetzt und die Validität der Thesen und Konzepte anhand von Beispielanwendungen verdeutlicht.

1.2 Abgrenzung

Die Inhalte der vorliegenden Arbeit stehen im Bezug zu diversen Forschungsgebieten und -themen. Dies erfordert eine Abgrenzung zu peripheren Forschungsaspekten, die im Rahmen dieser Arbeit nicht betrachtet werden können.

- Die Modellierung interaktiver Mashups stellt den Ausgangspunkt der Arbeit dar und erfolgt auf Basis des zu entwickelnden Metamodells. Das vorgelagerte Vorgehensmodell und etwaige Autorenwerkzeuge sind nicht Gegenstand der Arbeit, werden aber – ausgehend von Geschäftsprozessen – im Projekt DEMISA [@DEMISA, 2011] entwickelt. Dementsprechend wird bei der Komposition zumindest von Domänenexperten ausgegangen, während sich das angelagerte Projekt EDYRA [@EDYRA, 2011] der Frage widmet, wie Endnutzer auf Basis der geschaffenen Modelle selbständig Anwendungen erstellen können.
- Die Konzeption des Metamodells orientiert sich an den spezifischen Anforderungen und Charakteristika von Kompositionen unter Einbeziehung der Präsentationsebene. Dies schließt die An- und Einbindung verteilter Dienste ein. Es sollen aber keine komplexen Geschäftsprozesse und Transaktionen im Modell repräsentiert werden. Dafür wird auf bestehende Standards (BPEL) und Plattformen aufgebaut, die ihre Funktionalität i. d. R. über eine Service-Schnittstelle zur Verfügung stellen und darüber integriert werden können.
- Den Prinzipien des Component Based Software Engineering (CBSE) folgend, stellen Komponenten bzw. Dienste „Black Boxes“ dar, deren interne Funktionalität nur über ihre Schnittstelle beeinflusst werden kann. Die Adaptionskonzepte beschränkten sich folglich auf die Schnittstellenebene, wobei zur Entwicklung selbstadaptiver Mashup-Komponenten auf Ergebnisse verwandter Forschungsprojekte, wie AMACONT [Hinz, 2008] und HyperAdapt [@HyperAdapt, 2011], zurückgegriffen werden kann.
- Die Anwendung des Dienstprinzips auf die Präsentationsebene und die Komposition durch Nicht-Programmierer impliziert die Notwendigkeit neuer Mechanismen und Rahmenbedingungen zur Sicherstellung einer gewissen Dienstgüte und -verfügbarkeit. Auch wenn Qualitätsmaße zur Auswahl von Diensten beitragen, steht die Entwicklung von Qualitätssicherungsmaßnahmen und Testverfahren nicht im Fokus der Arbeit.
- Da sich die Arbeit der Präsentationsschicht von Webanwendungen widmet, ist der Bezug zu den Bereichen der *Human Computer Interaction*, *Visualisierung* und *Usability* gegeben. Die UI-Gestaltung stellt ein wichtiges Element für den Erfolg von Webanwendungen dar, liegt bei der Entwicklung von Mashups allerdings in der Verantwortung der Komponentenentwickler. Komponentenübergreifende Gestaltungsaspekte, wie das homogene *Look-and-Feel*, werden jedoch bei den in dieser Arbeit vorgestellten Konzepten berücksichtigt.
- Die Einbeziehung von Kontextinformationen in den Kompositionsprozess und die Ausführung von Anwendungen macht Mechanismen zur Kontextverwaltung nötig. Diese sollten von der adaptiven Anwendung entkoppelt werden, und eigenständig der Konsistenzsicherung und Ableitung neuen Kontextwissens für verschiedenste Anwendungen und Plattformen dienen. Die vorliegende Arbeit stützt sich dazu u. a. auf parallel entwickelte Ergebnisse.

- Das angestrebte Konzept geht von einer integrierten, nicht-verteilten Anwendung aus, mit der Nutzer über einen Client interagieren. Wie bei Webanwendungen üblich, können dazu einige Bestandteile auf dem Server ausgeführt werden. Eine weitere Verteilung i. S. v. *Distributed User Interfaces* oder Skalierungslösungen auf der Serverseite sind zunächst nicht vorgesehen. Ebenso wenig soll sich die Adaption auf die Systemarchitektur selbst erstrecken, z. B. in Form der dynamischen Anwendungsmigration.

1.3 Aufbau der Arbeit

Anhand der Zielstellung wird deutlich, dass die vorliegende Arbeit mehrere konzeptionelle Fragestellungen im Zusammenhang mit der modellgetriebenen Entwicklung adaptiver kompositer Anwendungen adressiert. Sie gliedert sich in acht Kapitel, deren Inhalte und Schwerpunkte sich an den drei genannten Schwerpunkten orientieren:

Kapitel 2 – Grundlagen, Szenarien und Herausforderungen
Das folgende Kapitel widmet sich der thematischen Einführung sowie der Klärung begrifflicher und konzeptioneller Grundlagen der Arbeit. Zur Veranschaulichung der Potentiale sowie der konzeptionellen und praktischen Herausforderungen werden exemplarisch drei Anwendungsszenarien vorgestellt und diskutiert. Ausgehend von diesen Erkenntnissen werden funktionale und nicht-funktionale Anforderungen an das Konzept formuliert, die gleichzeitig die Kriterien zur Analyse verwandter Arbeiten im darauffolgenden Kapitel bilden.

Kapitel 3 – Stand der Forschung und Technik
Kapitel 3 widmet sich der Darstellung und Analyse verwandter Arbeiten aus Forschung und Technik. Die Annäherung an die existierenden Konzepte erfolgt aus zwei verschiedenen Richtungen: einerseits werden Ansätze der SOA bzw. Dienstkomposition, andererseits Lösungen aus den Bereichen des Web- und Mashup-Engineering im Hinblick auf die gestellten Ziele untersucht. Die Diskussion der Probleme und Defizite hinsichtlich der in Kapitel 2 formulierten Anforderungen schließt das Kapitel ab und bildet die Überleitung zur Darstellung des neuen Konzepts.

Kapitel 4 – Universelle Komposition adaptiver Webanwendungen
In Kapitel 4 wird zunächst ein Überblick über das neue, wissenschaftliche Konzept zur modellgetriebenen Entwicklung adaptiver, kompositer Webanwendungen gegeben. Ausgehend von dem dabei unterstellten Rollenmodell werden die Schwerpunkte der Arbeit entsprechend den drei o. g. Herausforderungen erläutert und ihre Zusammenhänge dargelegt. Auf die Teilkonzepte wird in den folgenden Kapiteln genauer eingegangen.

Kapitel 5 – Belangorientierte Modellierung adaptiver, kompositer Webanwendungen
Kapitel 5 widmet sich dem Entwicklungsprozess adaptiver, komponentenbasierter Mashup-Anwendungen mit dem Schwerpunkt der Modellierung. Ein neuartiges, universelles Komponentenmodell bildet den Ausgangspunkt für die Vorstellung eines belangorientierten Metamodells zur plattformunabhängigen Repräsentation kompositer Anwendungen. Zur Unterstützung adaptiven Verhaltens zur Laufzeit wird danach auf dessen abstrakte Spezifikation in Form eines unabhängigen Teilmodells eingegangen.

Kapitel 6 – Kontextsensitiver Integrationsprozess und Kompositionsinfrastruktur

Gegenstand von Kapitel 6 ist die Konzeption einer verteilten Kompositionsarchitektur zur Unterstützung des modellbasierten Entwicklungsprozesses zur Laufzeit. Dazu wird eine innovative, serviceorientierte Infrastruktur vorgestellt, die die Verwaltung, dynamische Suche, Auswahl und Integration von Komponenten zu einer kompositen Webanwendung auf der Grundlage des o. g. Modells erlauben. Für deren Ausführung wird eine neuartige Referenzarchitektur vorgestellt, die auf verschiedene technologische Plattformen abgebildet werden kann. Schließlich werden Konzepte zur Unterstützung des modellierten, kontextadaptiven Verhaltens erläutert.

Kapitel 7 – Umsetzung und Validierung der Konzepte

Das siebente Kapitel beschreibt die praktische Realisierung und Validierung der geschaffenen Konzepte. Dabei wird sowohl auf die Formalisierung der entwickelten Modelle eingegangen als auch auf die technische Umsetzung der o. g. Kompositionsinfrastruktur und Ausführungsumgebungen. Die Darstellung verschiedener, umgesetzter Beispielanwendungen bietet einen Einblick in die praktische Nutzung und Validierung der Konzepte.

Kapitel 8 – Zusammenfassung, Diskussion und Ausblick

Im letzten Kapitel werden die Ergebnisse bzw. wissenschaftlichen Beiträge der Arbeit zusammengefasst und diskutiert. Grundlage für die Bewertung bilden die beschriebenen Probleme, Thesen und Forschungsziele. Abschließend wird ein Ausblick auf mögliche, zukünftige Arbeiten gegeben, die aufgrund der Abgrenzung und Fokussierung in dieser Arbeit ausgeklammert wurden.

2

Grundlagen, Szenarien und Herausforderungen

Im letzten Kapitel wurden die Ziele der Arbeit herausgearbeitet und diverse Forschungsfragen aufgeworfen, die mit der Modellierung und Ausführung dienstbasierter, interaktiver Anwendungen einhergehen. Sie betreffen insbesondere die plattformunabhängige Entwicklung und Kontextadaptivität vor dem Hintergrund der geschilderten Veränderungen im Web.
Die angestrebten Ziele implizieren die Nutzung, Weiterentwicklung und Kombination von Konzepten aus verschiedenen Forschungsgebieten. Zum einen sollen die Prinzipien der modellgetriebenen Softwareentwicklung für die strukturierte Entwicklung adaptiver, komponentenbasierter Webanwendungen eingesetzt werden, wobei verteilte Web-Ressourcen und -Dienste nach dem Vorbild der SOA als Anwendungsbausteine dienen. Zum anderen besteht ein Ziel dieser Arbeit in der Komposition und Integration der Anwendung zur Laufzeit, wodurch Aspekte der Daten- und Anwendungsintegration sowie der dynamischen Servicekomposition tangiert werden. Schließlich stellt die Einbeziehung von Kontext in den Kompositions- und Ausführungsprozess den Bezug zum großen Feld der kontextadaptiven Systeme und speziell zum *Adaptive Hypermedia* [Brusilovsky, 2001] her.
Um die Forschungsfragen adäquat beantworten zu können, bedarf es zunächst der begrifflichen und fachlichen Einordnung. In diesem Kapitel werden deshalb wichtige Begrifflichkeiten der Arbeit geklärt sowie die Charakteristika und konzeptionellen Grundlagen der angestrebten Lösungen beschrieben. Die Diskussion erfolgt anhand einiger Szenarien, die die praktischen Herausforderungen veranschaulichen und Referenzanwendungen für die spätere Evaluation in Abschnitt 7.3 darstellen. Das Kapitel schließt mit einem Überblick über die wesentlichen Anforderungen an das Konzept, die bei der späteren Betrachtung des Standes der Forschung und Technik als Bewertungskriterien dienen.

2.1 Grundlagen und Begriffsklärung

Für eine genaue Auseinandersetzung mit dem Thema und den verwandten Arbeiten müssen zunächst die grundlegenden Terminologien der formulierten Ziele geklärt werden. Die folgenden Abschnitte geben deshalb einen Überblick über die thematischen und begrifflichen Grundlagen der Arbeit und ordnen diese in den Forschungskontext ein. Ausgehend von der Vorstellung des hier unterstellten Komponentenbegriffs werden komponenten- und dienstbasierte sowie komposite Anwendungen begrifflich definiert. Daraufhin wird der Bezug zu Mashups hergestellt. Vor dem Hintergrund der angestrebten plattformunabhängigen Entwicklung widmet sich der darauf folgende Abschnitt den Begrifflichkeiten und Prinzipien der modellgetriebenen Entwicklung. Abschließend wird auf den Aspekt der Kontextadaptivität genauer eingegangen.

2.1.1 Komposite und serviceorientierte Webanwendungen

Nicht zuletzt aufgrund der Kosten- und Zeitersparnis bei der Softwareentwicklung haben sich Konzepte zur Steigerung der Wiederverwendbarkeit von Quellcode bzw. Software in den letzten Jahren stetig weiterentwickelt und an Granularität gewonnen. In der frühen funktionalen Programmierung erfolgte Wiederverwendung auf der Ebene von Funktionen. Mit dem Aufkommen objektorientierter Programmierung wurde die mehrfache Nutzung bzw. Instanziierung von Klassen und Objekten möglich. Komponenten- und dienstbasierte Systeme erlauben die Entwicklung und Bereitstellung lose gekoppelter „Bausteine“ noch höherer Granularität.

Komponentenbasierte Anwendungen

Der dieser Arbeit zugrunde liegende Komponentenbegriff folgt der prominenten Definition von Szyperski (2002), die für Anwendungsbestandteile folgende Charakteristika festlegt: Komponenten sind binäre Einheiten eines ausführbaren Systems, die unabhängig voneinander entwickelt, ausgeführt und bereitgestellt werden können. Sie sind kontextfrei und beliebig oft wiederverwendbar. Sie können durch Dritte komponiert werden, wobei Zugriff und Konfiguration im Sinne des jeweiligen Anwendungskontextes allein über vertraglich festgelegte Schnittstellen erfolgen (*Black-Box*-Prinzip).

Komponentenbasierte Anwendungen entsprechen diesen ausführbaren Systemen. Im Gegensatz zur objektorientierten Programmierung, die hauptsächlich Konzepte und ihre Zusammenhänge vor dem Hintergrund eines einheitlichen mentalen Modells modelliert, ist das Ziel des CBSE die funktionale Komposition neuer Anwendung aus vorgefertigten Bausteinen. Letztere können über ihre Schnittstellen an den Anwendungskontext angepasst und miteinander verknüpft werden.

Serviceorientierte Anwendungen

Die Entwicklung von Anwendungen unter Nutzung externer, verteilter Komponenten wird als Service Oriented Software Engineering (SOSE) bezeichnet, deren bekannteste Umsetzung die SOA ist. Diese stellt einen Entwicklungsrahmen dar, der es Entwicklern erlaubt, komplexe Anwendungen aus kleineren, verteilten Modulen bzw. Diensten zu erstellen [Tilsner et al., 2009]. SOA ermöglicht somit die Umsetzung komponentenbasierter Systeme auf Basis von Web Services.

Die dienstbasierte Entwicklung wird als logischer Nachfolger des komponentenbasierten Ansatzes verstanden, wobei sie insbesondere die Trennung von Schnittstelle und Implementierung über Anwendungs- und Unternehmensgrenzen hinweg betont. Kernidee ist die Verfügbarkeit verschiedenster verteilter Implementierungen für eine Spezifikation, deren Auswahl jederzeit, also ggf. erst zur Laufzeit, erfolgen kann [Estublier et al., 2010].

In diesem Zusammenhang und im Hinblick auf den im letzten Kapitel skizzierten Wandel von Internet und IT-Welt hat der Begriff *Cloud Computing* an Bedeutung gewonnen. Nach der verbreiteten Definition des National Institute of Standards and Technology (NIST) beschreibt dieser ein Modell für den einfachen Zugriff auf eine Menge geteilter Netzwerkressourcen im Bedarfsfall [Mell und Grance, 2011]. Durch Cloud Computing wird die Idee des SOC letztlich umfassend angewendet, sodass die verteilten Ressourcen Dienstleistungen auf verschiedensten Ebenen, einschließlich der Infrastruktur (Prozessorleistung, Speicherplatz, Virtualisierung, etc.) erbringen, wie in Abbildung 2.1 veranschaulicht wird.

Die in Abschnitt 1.1.3 vorgestellten Forschungsziele, insbesondere das angestrebte Kompositionsmodell und die Kompositionsinfrastruktur, sind darin der Platform as a Service (PaaS)-Schicht zuzuordnen, da sie Grundlage für Modellierung und Ausführung kompositer Anwendungen darstellen. Im Hintergrund kann durch die Laufzeitumgebung freilich auf auch auf Infastrukturdienste zugegriffen werden.

Konkrete Modellinstanzen repräsentieren mit dem Ansatz entwickelte, komponentenbasierte Anwendungen und sind somit auf der SaaS-Ebene angesiedelt. Gleiches gilt für die Komponenten bzw. Dienste, die miteinander verwoben werden. Selbstverständlich können diese intern auf Daten oder Infrastrukturdienste der Cloud zugreifen, was dem Anwendungsentwickler aufgrund des *Black-Box*-Prinzips jedoch verborgen bleibt. Die Vorteile von SaaS gegenüber konventioneller Software liegen auf der Hand: die weltweite Verteilung der Anwendung erfolgt nahezu kostenlos und Weiterentwicklungen können – ohne Kompilierung, Installation o. ä.– in kürzester Zeit bereitgestellt werden [Mikkonen und Taivalsaari, 2010].

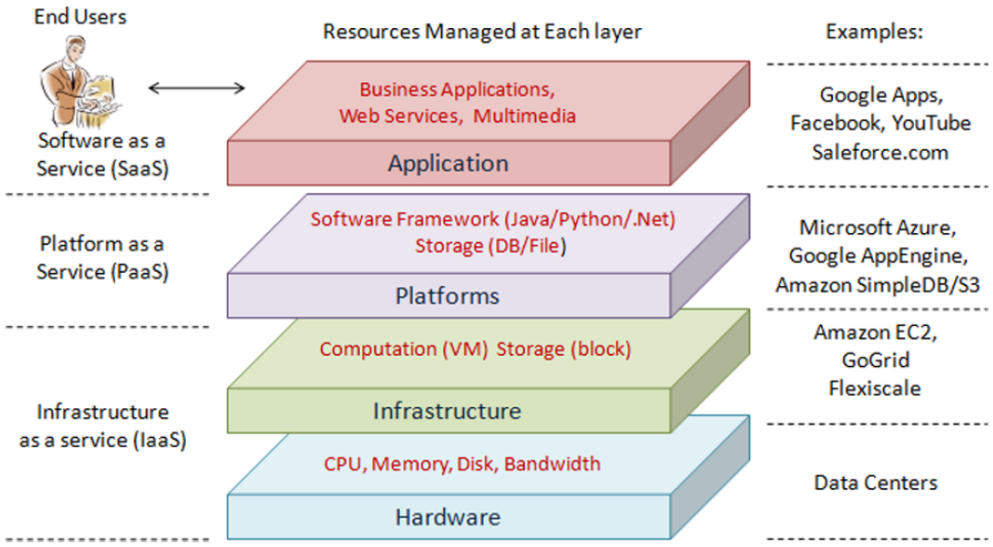

Abb. 2.1: Cloud Computing Stack [Zhang et al., 2010]

Komposite (Web-)Anwendungen und RIAs

Komposite Webanwendungen werden in der Literatur verschieden charakterisiert. Im Rahmen dieser Arbeit wird darunter eine Komposition von Web Services verstanden, die um eine Benutzerschnittstelle angereichert ist [Kapitsaki et al., 2008] bzw. deren Dienste durch einen *User-Interface-Flow* verbunden sind [Kateros et al., 2008].
Die Komposition von Diensten und Dienstleistungen erfolgt entsprechend den Prinzipien der SOA. Die resultierenden Anwendungen bzw. Modelle sind i. d. R. hochgradig formalisiert und stellen hohe Anforderungen an Entwickler und zugrunde liegende Infrastrukturen [Liu et al., 2007], was nicht zuletzt mit ihrer Ausrichtung auf das Geschäftsfeld korreliert. Die SOA-Prinzipien der Komponentisierung samt Autonomie, Entkopplung und Verteilung finden auf der Präsentationsebene kompositer Webanwendungen allerdings keine Anwendung. Für die Erstellung der entsprechenden Benutzerschnittstellen und die Abbildung von Nutzerinteraktionen kommen RIA-Oberflächen zum Einsatz, die manuell entwickelt werden müssen.
Diese Charakteristika widersprechen den angestrebten Zielen der vorliegenden Arbeit. Komposite Webanwendungen nach dem gängigen Verständnis bilden folglich nicht den Schwerpunkt der Konzeption. Vielmehr sollen die Beschränkungen auf SOAP-Dienste sowie auf die Daten- und Anwendungsebene überwunden werden.
Wenn später im Rahmen der Konzeption von kompositen Anwendungen gesprochen wird, so sind demnach *universelle* Kompositionen gemeint, die u. a. die manuelle, programmatische Entwicklung entsprechender RIA-Oberflächen unnötig macht.
Wird im Kontext der Konzeption von kompositen Anwendungen gesprochen, so wird demnach eine *universelle* Komposition unterstellt, die u. a. die manuelle, programmatische Entwicklung entsprechender RIA-Oberflächen unnötig macht.

Auch für den Begriff RIA gilt keine standardisierte oder zumindest klare Definition. Gemeinhin werden darunter Webanwendungen verstanden, die sich hinsichtlich der angebotenen Interaktionsmöglichkeiten und Benutzerschnittstellen von klassischen Hypermedia-Seiten abheben und eher Desktop-Anwendungen ähneln [Fraternali et al., 2010]. RIA implizieren keine Technologien oder Anwendungsmodelle, sondern sind durch die „Reichhaltigkeit“ und Dynamik ihrer Oberflächen charakterisiert, die durch verschiedenste Mittel – häufig kommt beispielsweise Ajax zum Einsatz – erreicht werden. Typische Merkmale sind die Unterstützung von *Drag-and-Drop*, die Steuerung über Tastenkürzel und die Autovervollständigung.
Mashups können als RIA aufgefasst werden, sofern ihre Oberflächen die beschriebenen Charakteristika vorweisen. Bei einer RIA handelt es sich im Umkehrschluss jedoch nicht zwingend um ein Mashup, da ihre Entwicklung i. d. R. programmatisch und nach klassischen Vorgehensmodellen der Softwareentwicklung erfolgt.

2.1.2 Mashups

Der Terminus *Mashup* beschreibt sowohl einen spezifischen Entwicklungsprozess als auch die daraus entstehenden Anwendungen. Letztere zeichnen sich dadurch aus, dass sie durch die Kombination existierender Web-Ressourcen, d. h. durch die Verknüpfung verteilter Daten und APIs, einen Mehrwert für den Nutzer generieren [Benslimane et al., 2008]. Dieser Mehrwert ist nicht zwingend funktional, sondern häufig qualitativer Natur gemäß dem Prinzip „Das Ganze ist mehr als die Summe seiner Teile“ [Aristoteles, 350 B.C.].

Mashups dienen – wie komposite Webanwendungen – der Komposition von Diensten im Sinne des SOC, adressieren jedoch einige grundlegende Probleme von SOA [Liu et al., 2007]. Im Gegensatz zu kompositen (Web-)Anwendungen steht die einfache, kostengünstige und schnelle Entwicklung im Vordergrund, um eine breite Zielgruppe bis hin zu Endnutzern zu erschließen und auch die nutzergetriebene Entwicklung und Evolution von Software – das EUD – zu ermöglichen. Deshalb wird bewusst auf die komplexen Orchestrierungsmechanismen von SOA verzichtet und von einer „leichtgewichtigen" Entwicklung gesprochen, die im Hinblick auf verwendete Formalismen, benötigte Infrastrukturen und den Entwicklungsprozess im Ganzen stärker auf Nicht-Programmierer ausgerichtet ist.
Mashups bauen auf einigen elementaren SOA-Prinzipien auf [Koschmider et al., 2009]. Dazu zählen die Autonomie und Komponierbarkeit der Mashup-Bestandteile bzw. Dienste sowie die klare Trennung von Dienstschnittstelle und -implementierung. Sie ermöglichen die lose Kopplung und Wiederverwendbarkeit von Teilen der Anwendung sowie die Suche und dynamische Integration von Diensten, u. U. auf Basis semantischer Informationen.
Verwandte Forschungsansätze zur Integration, insbesondere der Enterprise Application Integration (EAI) und Enterprise Information Integration (EII), sind i. d. R. auf die Daten- und Anwendungsebene beschränkt. Auch die Mehrzahl existierender Mashup-Ansätze geht von der Komposition dieser Art aus. Wie an verschiedener Stelle konstatiert, verschiebt sich der Schwerpunkt der Forschung in diesem Bereich jedoch zunehmend hin zur Entwicklung grafischer Benutzerschnittstellen [Di Lorenzo et al., 2009; Koschmider et al., 2009]. Dies spiegelt sich in der Mashup-Definition von Yu et al. (2008) wider, die sie als Webanwendungen bezeichnen, die „Daten, Anwendungslogik und Benutzerschnittstellen existierender Anwendungen und Dienste integriert." Auch im Rahmen dieser Arbeit soll die Präsentationsebene als inhärenter Teil der Komposition angesehen werden.
Mashups unterstellen nicht generell eine strukturierte oder komponentenbasierte Entwicklung. Die Verknüpfung von Diensten bzw. Dienstleistungen erfolgt häufig programmatisch, was den Anforderungen des EUD diametral entgegensteht. Die Unterstützung von Nicht-Programmierern im Entwicklungsprozess erfordert Paradigmen und Modelle, die die nötige Abstraktion von Spezifika der Dienstleistungen und der technologischen Plattformen und Sprachen erlauben, sowie entsprechende Autorenwerkzeuge. Vor diesem Hintergrund gibt es verschiedene Bestrebungen aus Forschung und Industrie, offene und erweiterbare Komponenten- und Kompositionsmodelle für Mashups zu entwickeln (vgl. Abschnitt 3.2.2).

Beim Einsatz von Mashups im geschäftlichen Umfeld unter Berücksichtigung der damit verbundenen Anforderungen spricht man gemeinhin von *Enterprise Mashups*. Auch wenn keine verbindliche Definition vorliegt, lassen sich einige Charakteristika festhalten: Enterprise Mashups adressieren ein wohldefiniertes Geschäftsproblem von taktischer und strategischer Relevanz [Ketter et al., 2009], welches unter Nutzung der Mashup-Prinzipien gelöst werden soll. Durch die Integration mit Geschäftsumgebungen sind die Grenzen zur klassischen SOA und den zugrunde liegenden Geschäftsprozessen häufig fließend. Es gilt die Prämisse, durch Enterprise Mashups der schwergewichtigen SOA innerhalb einer Firma ein leichtgewichtiges „Gesicht" zu verleihen [Hoyer und M. Fischer, 2008], welches es Mitarbeitern erlaubt, ihre Geschäftsprozesse und Probleme möglichst einfach und schnell selbst auf

Anwendungen abzubilden. Häufig werden dazu (firmen-)interne Legacy-Systeme und Dienste mit externen Datenquellen integriert [Crupi und Warner, 2008].
Aufgrund der geschäftlichen Nutzung adressieren entsprechende Vertreter insbesondere Aspekte der Qualitätssicherung (Robustheit, Verfügbarkeit, Quality of Service (QoS)), Sicherheit (Authentifizierung, Autorisierung, Datenschutz) und *Governance*. Diese Aspekte stehen zunächst nicht im Fokus dieser Arbeit, können aber durch Erweiterungspunkte im Konzept später integriert werden.

2.1.3 Modellgetriebene Software-Entwicklung

Wie bereits angesprochen, wird in dieser Arbeit ein strukturierter, modellgetriebener Entwicklungsprozess für Mashup-Anwendungen angestrebt. Dieser, in der Literatur als *Model Driven Development (MDD)* bezeichnet, impliziert die Verwendung von Modellen und Generatoren zur Unterstützung der Softwareentwicklung. Kernidee ist die iterative Erzeugung lauffähiger Anwendungen aus formalen Modellen, wobei Konzepte der Metamodellierung und Code-Generierung zum Einsatz kommen. Der Fokus der Entwicklung liegt jederzeit auf der abstrakten Modellebene, womit die technologienahe Programmierung von Anwendungslogik entfällt. In der bekanntesten Umsetzung von MDD, der Model Driven Architecture (MDA) [@OMG, 2011], erfolgt die schrittweise Spezifikation, Transformation und Verfeinerung eines Anwendungsmodells von einem reinen Fachmodell über eine plattformunabhängige und plattformspezifische Repräsentation des Softwaresystems bis hin zum Quellcode. Die Abbildung auf die jeweils nächste Abstraktionsebene wird durch Transformatoren vorgenommen und bedarf ggf. nachträglicher manueller Verfeinerung.
Modelle stellen in diesem Zusammenhang die abstrakte Repräsentation und Spezifikation eines (Software-)Systems hinsichtlich bestimmter Belange wie Struktur, Funktion und Verhalten dar. Häufig werden derartige Aspekte in verschiedenen, möglichst unabhängigen Teilmodelle repräsentiert, um den Entwicklungs- und Wartungsprozess zu vereinfachen – man spricht dann von *belangorientierten Modellen*. Sie werden durch Domain Specific Languages (DSLs) spezifiziert, wobei es sich um Sprachen handelt, die eine allgemeingültige Syntax und Semantik zur Repräsentationen von spezifischen Konzepten und Verhalten einer Domäne beschreiben [Deursen et al., 2000]. Jedes Modell wird in einer solchen Meta- oder Objektsprache dargestellt und ist gleichzeitig Instanz eines Metamodells. Zur modellgetriebenen Entwicklung komponentenbasierter Mashups bedarf es folglich mindestens eines Metamodells, welches für alle Belange der universellen Komposition, wie Komponenten, Daten- und Kontrollfluss, Layout etc., geeignete Modellierungskonzepte bereitstellt.
Durch die direkte Verknüpfung der modellbasierten Spezifikation mit dem Anwendungscode sollen Portabilität, Interoperabilität und Wiederverwendbarkeit von Softwaresystemen verbessert werden. Diese Formalisierung von Fachwissen auf höherem Abstraktionsniveau kann zum einen einfacher durch Domänenexperten – i. d. R. Nicht-Programmierer – bewerkstelligt werden, was der Zielgruppe von Mashups und dieser Arbeit entgegenkommt. Zum anderen kann abstraktes Wissen unabhängig von Plattform- und Technologieaspekten wiederverwendet und erweitert werden. Der permanente, automatische Abgleich zwischen Modell und Anwendungscode verhindert Inkonsistenzen bei Veränderungen auf jeweils einer Seite, z. B. in Folge

von Wartungsarbeiten. Somit entsteht eine qualitativ höherwertige Software und Dokumentation, während Entwicklungszeit und -aufwand abnehmen.
Computer-Aided Software Engineering (CASE) verfolgt einen ähnlichen Ansatz, geht allerdings von der vollautomatischen Erstellung des Programmcodes aus fachlichen Modellen aus. Im Gegensatz zum MDD bauen CASE-Werkzeuge auf vorgegebenen Domänenarchitekturen, d. h. Modellierungssprachen und Plattformen, auf.
Die Anwendung modellgetriebener Entwicklungsprinzipien auf RIA und komposite Webanwendungen wird an verschiedener Stelle motiviert [Wright und Dietrich, 2008; Achilleos et al., 2011], da sie als adäquates Mittel angesehen wird, um wechselnden fachlichen Anforderungen, rapiden technologischen Weiterentwicklungen und der großen Heterogenität von Zielgruppen und Endgeräten zu begegnen. Für eine ausführliche Darstellung der Prinzipien, Vor- und Nachteile des MDD sei auf Stahl et al. (2007) verwiesen.
Für die Entwicklung universeller Mashup-Anwendungen bietet sich MDD an, da hier von unabhängigen Teilkomponenten ausgegangen werden kann, deren Kopplung mit entsprechendem Vokabular auch auf höherer Abstraktionsebene modelliert werden kann. Im Gegensatz zum gängigen Vorgehen, aus den Modellen Anwendungscode zu generieren, muss das Ziel allerdings darin bestehen, diese Modelle auf existierende Komponenten und Kompositionen abzubilden. Bislang mangelt es jedoch an entsprechenden Konzepten.

2.1.4 Kontext und kontextadaptive Webanwendungen

Unter dem Begriff der Adaptivität versteht man in der Softwaretechnologie die Anpassungsfähigkeit einer (Web-)Anwendung an sich ändernde Umstände. In Anbetracht der im letzten Kapitel beschriebenen Heterogenität von Endnutzern sowie Hardware- und Softwareplattformen, aber auch im Hinblick auf komponenten- und dienstbasierte Architekturen gewinnt die Adaptivität von Anwendungen immer mehr an Bedeutung. Die Verfügbarkeit und Verlässlichkeit aller Anwendungsbestandteile – auch jener entfernten und dynamisch gebundenen – muss bei wechselnden Umständen sichergestellt werden. Aus Gründen der Flüchtigkeit (Ausfälle, Wartungszyklen, Verbindungsabbrüche) und Evolution (Weiterentwicklung, Ersatz) von Diensten sowie schwankenden Nutzer- und Qualitätsanforderungen müssen sich Anwendungen stetig den aktuellen Gegebenheiten anpassen.
Zur Charakterisierung adaptiver Anwendungen können die gängigen W-Fragen herangezogen werden [Brusilovsky, 1996]:

Woran? Frühe Systeme konzentrierten sich auf die *Personalisierung* von Anwendungen – die Anpassung an den Nutzer und seine Eigenschaften. In zunehmendem Maße wurden vielfältige äußere Einflüsse einbezogen und der Begriff der „kontextadaptiven Anwendungen“ geprägt. Die prominenteste Definition von Dey und Abowd (1999) bezeichnet Kontext als „jegliche Information, die zur Charakterisierung der Situation einer Entität dienen kann. Eine Entität ist eine Person, ein Ort, oder ein Objekt, dass bezüglich der Interaktion zwischen Nutzer und Anwendung Relevanz besitzt und beide einschließt.“
Die zur Verfügung stehenden Kontextinformationen können nach Lizcano et al. (2008) vier Kategorien zugeordnet werden. Der Auslieferungskontext definiert die technische Umgebung der Anwendung zur Laufzeit, z. B. Eigenschaften vom

Endgerät oder die Netzwerkverbindung. Die zweite Kategorie beschreibt den Nutzer, dessen Identität, Charakteristika, Vorlieben, usw. Weiterhin können Kontextdaten der physischen Umgebung Anpassungen auslösen, z. B. der Ort der Nutzung, der Geräuschpegel oder die Helligkeit. Als letzte Kriterien werden die Situation und Zeit der Nutzung genannt, wie das aktuelle Wetter oder die Unterscheidung von beruflichem und privaten Einsatz. Da komposite Anwendungen aus verteilten Diensten zusammengesetzt werden, muss auch deren QoS bzw „Güte" zur Kompositions- und Laufzeit beachtet werden, um bei Verstößen adäquat reagieren zu können.

Wie? Für die Anpassung von Software existieren verschiedene Adaptionsmethoden und -techniken, die auf die jeweiligen Anwendungsarchitekturen zugeschnitten sind. Die Anpassungsmöglichkeiten serviceorientierter Systeme unterscheiden sich beispielsweise deutlich von den prominenten Adaptionskonzepten des *Adaptive Hypermedia* [Brusilovsky, 2001].

Wer? Urheber einer Anpassung kann der Entwickler, der Endnutzer oder die Software selbst sein. Bei der Adaption durch den Endanwender spricht man von der *Adaptierbarkeit* einer Anwendung. Geht die Initiative zur Anpassung vom System selbst aus – egal ob eigenständig oder vom Entwickler vordefiniert – spricht man von *Adaptivität*. Letztere steht im Fokus dieser Arbeit.

Was? Auch der von der Adaptionstechnik betroffene Bestandteil einer Anwendung hängt stark von der Art der selbigen ab. In Hypermedia-Systemen wird zwischen der Adaption von Inhalt, Präsentation, Navigation und Architektur unterschieden, während in komponenten- und dienstbasierten Systemen aufgrund der starken Kapselung der Bestandteile die Anpassung auf Anwendungsebene (innerhalb von Komponenten), auf Schnittstellenebene (Komponenten) und auf Kompositionsebene unterschieden werden kann.

Warum? Auslöser für Adaptionen können so vielfältig wie der Kontext selbst sein. Je nach Art der Anpassung können beliebige Kontextinformationen einschließlich Nutzerinteraktionen zur Anpassung führen (vgl. [Ketfi et al., 2002]). Überaus prominent ist die Nutzung des Ortes durch Location Based Services (LBS), wie bereits in einem der ersten kontextbewussten Systeme, den *Olivetti Active Badges* [Want et al., 1992]. Im SOA-Umfeld führt hingegen häufig die Sicherstellung bzw. Verletzung von QoS-Kriterien zur Adaption.

Wann? Die Entscheidung zur Anpassung kann zu verschiedenen Zeitpunkten erfolgen. Bereits beim Deployment können durch Konfiguration oder Code-Generatoren bestimmte Funktionalitäten beeinflusst werden. Kontextinformationen können weiterhin in den Kompositionsprozess, z. B. in die Auswahl und Bindung von Diensten, einbezogen werden. Schließlich müssen auch bereits komponierte Anwendungen zur Laufzeit an veränderliche Situationen angepasst werden können.

Unter einer kontextadaptiven Anwendung wird also jegliche Software verstanden, die unter Ausnutzung von Kontext bedarfsgerechte Dienste und Inhalte bietet, die sich selbständig an den Kontext anpasst, oder beides [Daniel, 2010].

2.2 Szenarien und Problemanalyse

Die Komposition unter Einbeziehung der Benutzerschnittstelle bringt eine Vielzahl von Herausforderungen für den Entwicklungsprozess und die Ausführungsumgebung mit sich. Die folgenden Anwendungsbeispiele dienen der Veranschaulichung und Konkretisierung der Probleme sowie der Ableitung von Anforderungen und Bewertungskriterien. Letztere bilden die Grundlage für die Einschätzung verwandter Ansätze im nächsten Kapitel.

2.2.1 Dienstkomposition zur Reiseplanung

Die Anwendung „TravelMash“ dient der Planung einer Reise zu einem frei wählbaren kulturellen Ereignis samt der Routenplanung und der Suche nach Hotels in der Nähe. Abbildung 2.2 zeigt die Komposition schematisch (links) und zur Laufzeit (rechts). Es wird deutlich, dass eine Reihe von Diensten bzw. Komponenten miteinander gekoppelt werden, die die benötigten Daten, Funktionalitäten und Oberflächen bereitstellen. Grundlage für die Reiseplanung bieten die Datendienste ⑦: Einer liefert für gegebene Orte passende Konzerte, Ausstellungen und sonstige Ereignisse. Ein anderer sucht Hotels, die in einem festen Umkreis des gegebenen Ortes verfügbar sind. Ein weiterer berechnet, ausgehend von Startzeit und -ort sowie Zielort, die günstigsten Verbindungen unter Nutzung des öffentlichen Nahverkehrs. Weitere Mehrwertdienste, z. B. zur Abfrage von Wetterdaten am Zielort, runden die Anwendung ab.

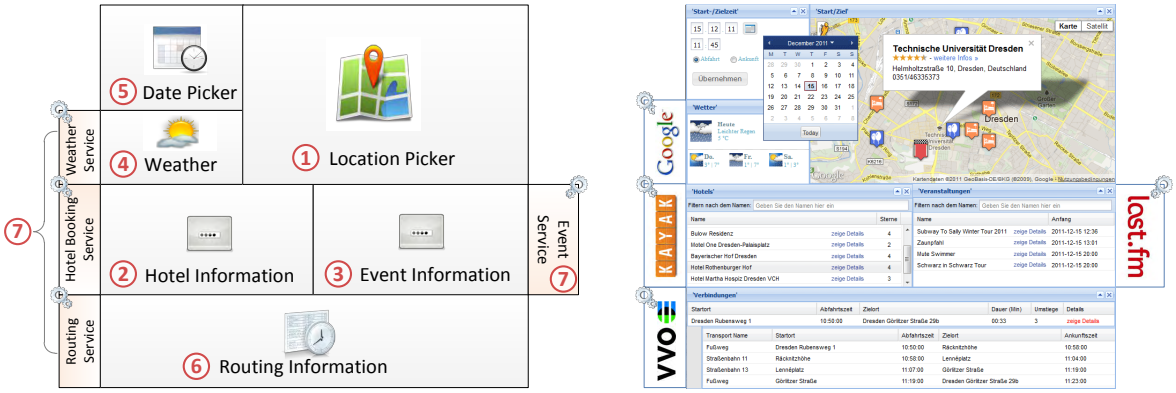

Abb. 2.2: Beispielkomposition zur Reiseplanung zur Design- und Laufzeit

Die Eingabe, Auswahl und Manipulation von Daten wird durch UI-Komponenten ermöglicht. Sie erlauben dem Nutzer, eine Reise wie folgt zu planen:

- Die Auswahl von Start- und Zielort der Reise erfolgt im Normalfall über eine Karte ①. Je nach verfügbarem Platz, installierten Plug-ins und Vorlieben des Nutzers, werden Karten verschiedener Anbieter genutzt. Bei Beschränkungen des Endgerätes kann die Auswahl auch textuell oder anhand vorgegebener Werte, z. B. durch ein Such- bzw. Formularfeld, erfolgen. Demzufolge müssen Komponenten abstrakt spezifiziert werden, um zwischen alternativen Implementierungen wählen zu können. Die Auswahl muss zur Initialisierungszeit der kompositen Anwendung in Abhängigkeit vom aktuellen Kontext erfolgen.
- Für ausgewählte Zielorte werden Ereignisse und Hotels in einem vorgegebenen Radius angezeigt, z. B. in sortierbaren Listen ② ③. Dazu müssen die Eingaben

auf der Benutzeroberfläche ① zunächst an die dahinter liegenden Dienste ⑦ weitergeleitet werden. Die resultierenden Ergebnisse, wie verfügbare Hotels, müssen gleichsam zur Oberfläche zurückgelangen. Dazu muss der Datenfluss zwischen Komponenten über Anwendungsebenen hinweg definiert werden. Eine Herausforderung dabei stellen die potentiell verschiedenen Datenmodelle der Komponenten dar, deren syntaktische Differenzen überbrückt werden müssen.

- Bei ausreichender Abstraktion der Schnittstelle kann für die Auflistung von Hotels ② und Events ③ dieselbe generische Komponente zum Einsatz kommen. Diese muss zur Laufzeit zweifach initialisiert und mit den verschiedenen Datenmodellen parametrisiert werden. Die Wiederverwendung trägt zur Zeit- und Kostenersparnis bei der UI-Erstellung bei.
- Zusätzlich zur Auflistung werden die verfügbaren Hotels ② und Ereignisse ③ auf der Karte ① angezeigt und sind dort als Ziele auswählbar. Die Instanzdaten müssen somit zwischen Komponenten (Karte ①, Listen ②,③ und Hintergrunddiensten ⑦) abgeglichen werden. Folglich müssen Zustandsänderungen von Komponenten nach außen publiziert werden, um die Konsistenz innerhalb der Anwendung sicherzustellen. Dabei ist es irrelevant, ob sie durch Nutzerinteraktionen (Auswahl von Orten auf der Karte), aktualisierte Daten (neue Wetterinformationen) oder durch Berechnungen hervorgerufen werden.
- Eine UI-Komponente ④ zeigt Wetterinformationen vom Zielort an. Hier kommen die o. g. Mechanismen zum Einsatz: Verändert sich der Zustand der Karte, z. B. ihr „Ziel-Marker“, durch Nutzerinteraktion, wird diese Änderung publiziert. Der Wetterdienst muss die bereitgestellten Daten verarbeiten und seinerseits Wetterdaten an die UI-Komponente weiterleiten.
- Nun kann der Nutzer die Start- oder Ankunftszeit seiner Reise wählen. Dazu stehen ihm UI-Elemente zur Auswahl von Zeit und Datum zur Verfügung ⑤, die allein auf der UI-Ebene verortet sind und von keinerlei Dienst abhängen.
- Sobald ein Event in der Liste ③ oder Karte ① ausgewählt wird, wird der Beginn der Veranstaltung automatisch als Ziel- bzw. Ankunftszeit gesetzt. Hierzu können die o. g. Publikationsmechanismen zum Einsatz kommen.
- Nach der Auswahl von Start- und Zielort in Form einer Adresse, eines Events oder Hotels, sowie der Start- oder Zielzeit, erfolgt die automatische Berechnung von Routen mit öffentlichen Verkehrsmitteln. Die Zustände der verschiedenen UI-Komponenten bilden in ihrer Gesamtheit die Eingabe für den Dienst zur Routenberechnung. Demzufolge muss neben der 1:1- und 1:n-Verknüpfung von Komponenten bzw. Diensten auch die n:m-Kopplung ermöglicht werden.
- Mögliche Routen werden für den Nutzer sortierbar aufgelistet ⑥. Die Auswahl einer Route liefert auf Wunsch Details wie Umsteigeinformationen.

Es zeigt sich, dass diese einfache Komposition bereits einige Anforderungen mit sich bringt, die in Abschnitt 2.3 und danach bei der Bewertung existierender Ansätze aufgegriffen werden. Der nächste Abschnitt problematisiert anhand eines weiteren Szenarios gesondert die Herausforderungen der Kommunikation zwischen bzw. Koordination von Komponenten.

2.2.2 Interaktive Aktienverwaltung

Eine weitere (vereinfachte) Anwendung – Abbildung 2.3 zeigt die Oberfläche von „StockMash“ – dient der Analyse und Verwaltung eines Aktiendepots. Zwei Dienste im Hintergrund bieten den Zugang zu Aktienindizes und -kursen sowie dem eigenen Depot. Die Visualisierung verfügbarer Aktien erfolgt einerseits in Listenform ①, andererseits gibt eine Detailansicht ④ genauere Auskunft, z. B. über die Kursentwicklung der letzten Wochen, Monate oder Jahre. Zwei weitere UI-Komponenten bieten die Möglichkeit, das eigene Depot einzusehen und ihm neue Aktien hinzuzufügen.

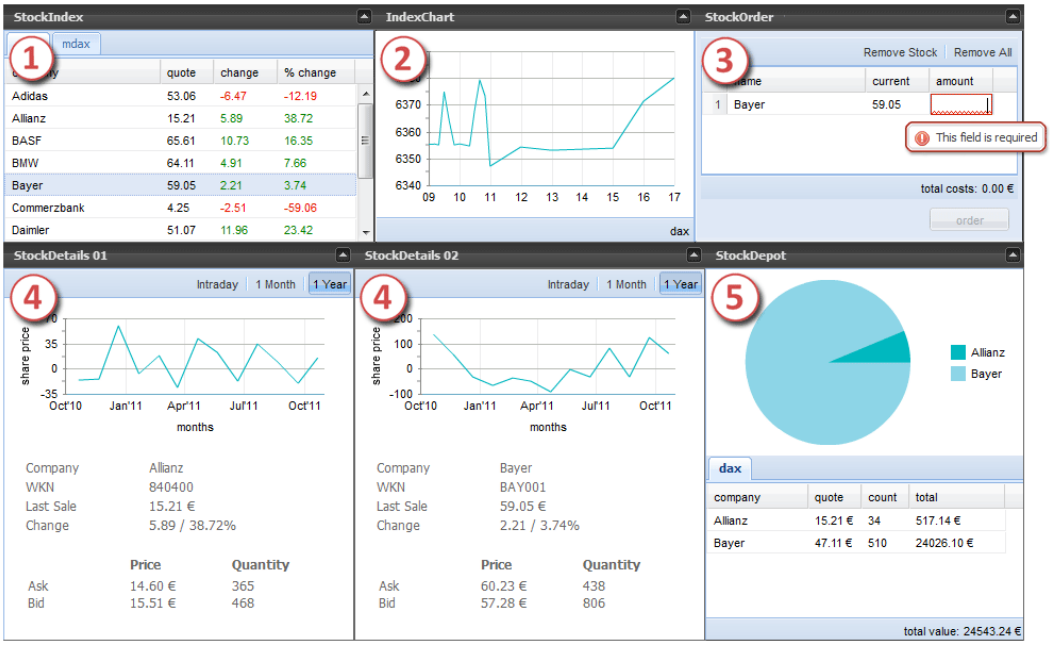

Abb. 2.3: Beispiel einer kompositen Anwendung zur Aktienverwaltung

Die Funktionalität der Anwendung lässt sich wie folgt zusammenfassen:

- Verfügbare Aktien werden aufgelistet ① und können u. a. nach Name und Kurs sortiert werden. Dazu greift die UI-Komponente auf eine Komponente im Hintergrund zurück, die zyklisch die aktuellen Kurse bereithält. Deshalb sollten Anfragen mit Rückantwort (*Request-Response*) zwischen Komponenten unterstützt werden, ohne das Prinzip der losen Kopplung aufzuweichen.
- Die Auswahl eines Aktienindizes (z. B. DAX, MDAX) zeigt dessen Verlauf in Diagrammform ② neben der Kursliste. Dies setzt die Weitergabe von Informationen zwischen Komponenten wie im Reiseplanungsszenario voraus.
- Aktien können mittels Drag-and-Drop aus der Übersicht ① in die Komponente zum Ordern gezogen werden ③, d. h. die Koordination erfolgt durch den Nutzer selbst. Dafür müssen u. a. Interaktionstechniken wie Drag-and-Drop durch die Laufzeitumgebungen unterstützt und auf das einheitliche Kommunikationsmodell abgebildet werden. Die Komponenten selbst sollten im Sinne der Wiederverwendbarkeit jedoch unabhängig von eingesetzten, plattformspezifischen Interaktionsformen bleiben.
- Nach der Angabe der Ordermenge ③ können Aktien in das Depot übernommen werden. Die Angabe einer Ordermenge ist obligatorisch, d. h. im Fehlerfall

wird der Nutzer darauf hingewiesen, wie in der Abbildung ersichtlich. Die Fehlerbehandlung muss durch die Orderkomponente ③ selbst erfolgen, da die Laufzeitumgebung kein Wissen über die internen Abläufe und UI-Elemente der integrierten Komponenten besitzt.

- Die Anzeige zum Aktiendepot ⑤ aktualisiert sich automatisch, falls eine Order ausgelöst wurde. Die UI-Komponenten kommunizieren dazu nicht untereinander, sondern indirekt über den Depot-Dienst. Das Auslösen der Order führt zum Dienstaufruf im Hintergrund, und die Änderung des Depots wiederum zur Benachrichtigung der entsprechenden UI ⑤. Folglich sind einheitliche Kommunikationsmechanismen über alle Anwendungsebenen hinweg nötig.
- Aktien können aus der Übersicht ① per Drag-and-Drop auf eine der Detailansichten ④ gezogen werden, um genauere Informationen über sie zu erhalten. Die Koordination kann im Gegensatz zum oben beschriebenen Fall nicht vormodelliert werden, da das Ziel der Daten aus Sicht der Aktienliste ① nicht eindeutig bestimmt ist. Ein Datenausgang für ausgewählte Aktien steht hier Eingängen von potentiell n Detailansichten gegenüber. Um zwei oder mehr Aktien miteinander vergleichen zu können, müssen die Daten an jeweils unterschiedliche Komponenten weitergeleitet werden.
- Zu den ausgewählten Aktien werden in der Detailansicht ④ Informationen von einem Dienst angefordert und fortwährend aktualisiert. Bezüglich der Anfrage benötigter Informationen von Diensten bestehen die gleichen Anforderungen, wie bei der Aktienliste ①. Um jedoch ein ständiges *Polling* seitens der UI zu vermeiden, sollte es die Möglichkeit geben, Kommunikationskanäle offen zu halten, sodass angefragte Komponenten in festgelegten Abständen selbständig Aktualisierungen an den Anfrager senden können.
- Beim Wechsel des in einer Detailansicht ④ sichtbaren Zeitraums des Kursverlaufs passen sich alle anderen Detailansichten an, damit die Vergleichbarkeit gegeben ist. Hierzu muss die Synchronisation zwischen Komponenten möglich sein – in diesem Fall hinsichtlich der Eigenschaft „Zeitraum“. Die Synchronisation zwischen Komponenten bzw. Eigenschaften muss somit modellierbar sein und auch zur Laufzeit unterstützt werden, wobei etwaige Inkonsistenzen und Zyklen erkannt und behandelt werden müssen.

Dieses Szenario zeigt anschaulich die Herausforderungen, die sich bei der universellen Komposition hinsichtlich der Kommunikation und Koordination von Komponenten ergeben. Insbesondere die Unterstützung verschiedener Kommunikations- und Synchronisationsmuster gewinnt durch die Einbeziehung der Präsentationsebene in Kompositionen an Relevanz. Auch diese Kriterien werden deshalb bei der Anforderungsanalyse aufgegriffen und bei der späteren Analyse verwandter Arbeiten angelegt. Das folgende Szenario illustriert schließlich die Herausforderungen im Bezug auf die Unterstützung adaptiven Verhaltens.

2.2.3 Adaptive Touristeninformation

Das letzte Anwendungsbeispiel widmet sich den vielfältigen Adaptionsmöglichkeiten in interaktiven Mashups. In diesem Zusammenhang soll ein Beispiel aus dem populären Anwendungsgebiet der LBS einen praktischen Eindruck der Möglichkeiten und Herausforderungen von Adaptivität geben.

Die adaptive Touristeninformation *TravelGuide* erlaubt die Suche nach Sehenswürdigkeiten, Restaurants und Hotels in der Umgebung des Nutzers, bindet dazu verschiedene Dienste an und visualisiert deren Daten auf einer Karte und einer Reihe weiterer UI-Komponenten. Für alle Orte können Detailinformationen in Form von Texten, Bildern und Videos sowie ein Routenplan eingeblendet werden. Mehrwertdienste, wie eine Wettervorhersage, runden die Anwendung ab. Die Praktikabilität und Nutzbarkeit von TravelGuide hängt in starkem Maße von der angemessenen Einbindung von Kontextwissen ab. Die folgenden Beispiele verdeutlichen dies:

- Alle in TravelGuide enthaltenen Komponenten nutzen die Muttersprache des Nutzers. Dies impliziert, dass die Konfiguration von Komponenten eines Mashups durch Kontextparameter beeinflusst wird, z. B. indem ihre Konfigurationsparameter mit Eigenschaften aus dem Kontextmodell verknüpft werden.
- Die Karte wird automatisch auf die aktuelle Position des Nutzers zentriert, die über GPS oder den Browser ermittelt wird. Die geplante Route passt sich fortwährend an, wenn sich der Nutzer bewegt. Bei veränderlichen Kontextparametern, wie dem Ort des Nutzers, ist somit nicht nur ein initialer Kontextbezug gegeben, sondern eine fortwährende Aktualisierung der Komponenten nötig.
- Die Auswahl der angezeigten Orte ist an die Interessen und Eigenschaften des Nutzers angepasst. Für Vegetarier werden nur entsprechende Restaurants angezeigt, für Kunstinteressierte die Museen in der näheren Umgebung. Die Karte zeigt nutzerspezifische Zusatzdaten, wie Bordsteininformationen für Gehbehinderte. Der Zustand von Komponenten ist somit direkt von Parametern des Kontextmodells abhängig, was u. a. die bereits erwähnte, ggf. dynamische (Re-)Konfiguration von Komponenten nötig macht.
- Die Routenplanung beachtet, ob sich der Nutzer zu Fuß oder mit dem Auto fortbewegt, und ob er über etwaige Behinderungen verfügt, die Umsteigezeiten beeinflussen könnten. Während die bislang erwähnten Beispiele auf die kontextabhängige Konfiguration von Komponenten bzw. Diensten abbildbar sind, ist hier ein Funktions- bzw. Methodenaufruf kontextabhängig zu gestalten.
- Die Anwendung passt sich an die verfügbare Darstellungsfläche an. So unterscheiden sind Layout und angebotene Informationen einer mobilen Variante von denen einer Desktop-Variante. Kontexteigenschaften haben also nicht nur Einfluss auf Komponenten selbst, sondern auf die Komposition als Ganzes. Kontextabhängige Änderungen der Komposition, z. B. hinsichtlich des Layouts, des Datenflusses, unterschiedlicher „Sichten“, etc., müssen abstrakt formulierbar sein und zur Laufzeit umgesetzt werden.
- Die Eingabe- und Interaktionsmöglichkeiten der integrierten UI-Komponenten sind an das Endgerät angepasst. Sehenswürdigkeiten können – je nach Situation und Endgerät – durch Maus-, Touch- oder Spracheingabe ausgewählt werden. Bei der Modellierung muss davon abstrahiert werden, welche Implementierung zur Laufzeit zum Einsatz kommt. Diese Entscheidung muss bei der Initialisierung der Anwendung auf Basis der verfügbaren Informationen über Endgerät und Nutzer getroffen werden, was einen kontextsensitiven Auswahl- und Integrationsprozess für Komponenten impliziert.
- Adaptionsmöglichkeiten erlauben die Minimierung, Entfernung und den Austausch von Komponenten. Neben der Atomarität und Isolation muss beim

Komponententausch insbesondere der Zustandserhalt gewährleistet werden. Der Zustand einer Komponente – z. B. die auf einer Karte sichtbaren Marker – muss von der ursprünglichen auf die neu einzufügende Komponente übertragen werden, um die Konsistenz innerhalb der Mashup-Anwendung sicherzustellen.

- Bei der Nutzung eines mobilen Endgeräts führt ein niedriger Batteriestand zum Austausch rechenintensiver Komponenten, z. B. eines Video-Players durch eine Diashow. Gleichsam erfolgt beim Ausfall eines Dienstes oder der Verletzung von QoS-Anforderungen die Ersetzung der betroffenen Komponente mit einer alternativen Implementierung. Auch hierzu muss Adaptionsverhalten in Abhängigkeit von Kontextparametern abstrakt formulierbar sein und die o. g. Techniken (Rekonfiguration, Austausch, etc.) beinhalten.

Neben der systemgetriebenen Adaption hat auch der Nutzer die Möglichkeit, sich die Anwendung an seine Anforderungen anzupassen. So kann er Komponenten verschieben, sie minimieren, wiederherstellen und auch entfernen. Entfernte Komponenten können zu einem späteren Zeitpunkt wieder eingefügt werden. Auch die Beeinflussung der Komposition selbst im Sinne des EUD ist denkbar, steht allerdings nicht im Fokus dieser Arbeit.

2.3 Anforderungen und Kriterien der Analyse

In den vorhergehenden Teilkapiteln wurden die grundlegenden Begrifflichkeiten der Arbeit geklärt und die damit verbundenen Charakteristika und Prinzipien vorgestellt. Anhand von drei Beispielszenarien wurden die Probleme und Herausforderungen der universellen Komposition hinsichtlich der Modellierung und Ausführung praktisch verdeutlicht. Im Folgenden werden daraus die wichtigsten Anforderungen an die zu entwickelnden Konzepte abgeleitet. Sie sollen sowohl der kritischen Betrachtung und Bewertung verwandter Arbeiten als auch als Leitfaden für die darauf folgende Konzeption dienen. Für die spätere Zuordnung und Rückverweise werden sie jeweils mit einem Stichwort versehen.

2.3.1 Anforderungen an Komponenten- und Kompositionsmodell

Um die Entwicklung interaktiver kompositer Anwendungen für Nicht-Programmierer zu unterstützen, bedarf es Modellierungsmitteln bzw. -konzepten, die einem einheitlichen Kompositionsparadigma unterliegen.

A. Komponentenmodell

Die wichtigsten mit der Modellierung und Beschreibung von Komponenten verbundenen Anforderungen lauten:

Komponentenorientierung Verteilte Web-Ressourcen soll als Komponenten repräsentiert und den Prinzipien komponenten- und dienstbasierter Ansätze entsprechen, d. h. alle Anwendungsbestandteile müssen modular, parametrisierbar und über standardisierte Schnittstellen ansprechbar sein (vgl. Assmann (2003)). Im Sinne der Wiederverwendbarkeit sollten keine expliziten, funktionalen Abhängigkeiten zwischen Komponenten bestehen, um eine lose Kopplung und größtmögliche Flexibilität der Anwendungen zu unterstützen.

Universalität Alle Komponenten, egal ob im Frontend oder Backend angesiedelt, sollen anhand einheitlicher Paradigmen beschreib- und komponierbar sein. Dabei sollen insbesondere die Anforderungen aus UI-Sicht einbezogen werden, die u. a. ein zustandsbehaftetes Modell nötig machen.

Beschreibung Die Suche und Integration von Komponenten soll auf Basis einer generischen, technologieunabhängigen Beschreibung erfolgen, die sowohl einfach und menschenlesbar als auch maschinell verarbeitbar ist. Der genutzte Formalisierungsgrad der Komponentenbeschreibung sollte ein Kompromiss aus Ausdrucksfähigkeit bzw. Komplexität und Aufwand der Erstellung sein. Zudem ist auf die einfache Erweiterbarkeit des Formates Wert zu legen.

Semantik Um die kontextadaptive Auswahl und Konfiguration von Komponenten zu erlauben, sollte die Beschreibungssprache von syntaktischen Eigenheiten abstrahieren und die semantische Auszeichnung unterstützen. Letztere sollte sich nicht auf die Typisierung der ausgetauschten Daten beschränken, sondern auch funktionale und nicht-funktionale Kriterien abdecken, um die Ausdrucksfähigkeit und die Präzision bei der späteren Suche zu steigern.

B. Kompositionsmodell

Die Modellierung interaktiver Anwendungen auf Basis dieser Komponenten muss sich den folgenden Anforderungen stellen:

Abstraktion Das Modell muss die Basiskonzepte einer Komposition in Form der o. g. Komponenten repräsentieren und deren anwendungsspezifische Konfiguration erlauben. Im Sinne der kontextsensitiven Bindung müssen Abstraktionsmöglichkeiten existieren, um Eigenheiten der Implementierungen oder verwendeten Technologien ausblenden zu können. Dies kann in Form funktionaler und nicht-funktionaler Anforderungen geschehen, die zur Laufzeit mit den Eigenschaften verfügbarer Komponenten abgeglichen werden.

Koordination Das Modell muss den Daten- und Kontrollfluss der Anwendung abstrakt beschreiben, indem die integrierten Komponenten miteinander in Beziehung gesetzt werden. Dabei sollten die spezifischen Kommunikations- und Koordinationsformen interaktiver Komponenten beachtet werden, z. B. die Synchronisation von Komponentenzuständen (vgl. Detailansichten der Aktienverwaltung), aber auch über Anwendungsebenen hinweg (Synchronisation von Aktiendepot und Depotansicht) sowie die Unterstützung reichhaltiger Interaktionsformen wie Drag-and-Drop.

Interoperabilität Die unabhängige Entwicklung und angestrebte lose Kopplung von Komponenten geht einher mit potentiellen Differenzen der verwendeten Datenmodelle und Schnittstellen. Zur Steigerung der Interoperabilität müssen deshalb Modellierungsmittel zur Überbrückung derartiger syntaktischer Unterschiede auf der Signatur- und der Datenebene bereitgestellt werden.

Präsentation Im Gegensatz zur konventionellen Dienstkomposition werden Modellkonstrukte zur Definition von Formatierungseigenschaften und Layouts der Anwendung benötigt, um das visuelle Erscheinungsbild beeinflussen und vereinheitlichen zu können. Verschiedene Sichten, z. B. zur Abbildung sequentieller Dialogschritte, sollten ebenfalls modellierbar sein.

Kontextsensitivität Um die Anpassung einer Mashup-Anwendung zur Laufzeit zu ermöglichen, müssen adaptives Laufzeitverhalten und Kontextbedingungen modellierbar sein. Zum einen ist es notwendig, Komponenten direkt mit Kontexteigenschaften zu verknüpfen. Zum anderen sollte sich die Adaption nicht auf einzelne Komponenten beschränken, sondern auch Änderungen auf Kompositionsebene, z. B. des Layouts oder Datenflusses, unterstützen.

Plattformunabhängigkeit Die Beschreibung einer Anwendung sollte von Spezifika der eingesetzten Technologien und Ausführungsplattformen abstrahieren, um die in Abschnitt 2.1.3 geschilderten Vorteile des MDD für interaktive Web- bzw. Mashup-Anwendungen nutzen zu können.

Erweiterbarkeit Die leichte Erweiterbarkeit einer Anwendung soll durch das Modell gewährleistet sein: Das Hinzufügen einer neuen Komponente muss allein auf Basis ihrer abstrakten Schnittstellenbeschreibung und ohne Programmierung möglich sein. Auch das Metamodell selbst sollte leicht erweiterbar sein, um später weitere Anwendungsaspekte modellieren zu können.

Wiederverwendbarkeit Neben der mehrfachen Einbindung von Komponenten in eine Anwendung – wie im Aktiendepot-Beispiel – oder über Anwendungsgrenzen hinweg, kann die Wiederverwendung und Skalierbarkeit über die rekursive Komposition von Komponenten gesteigert werden. Das Kompositionsmodell sollte dementsprechend die Hierarchisierung von Kompositionen, d. h. die Nutzung von Kompositionen als neue Komponenten, ermöglichen.

Neben den obigen Kriterien müssen eine Reihe weiterer, nicht-funktionaler Anforderungen berücksichtigt werden, die sich aus den in Kapitel 1 geschilderten Zielen und der Zielgruppe ergeben. Dazu zählt u. a. die Unterstützung der **Separation of Concerns (SoC)**, indem insbesondere der Aspekt der Kontextadaptivität von den restlichen Belangen der Anwendungen entkoppelt wird. Zudem sollte im Erstellungsprozess auf **Standardkonformität** geachtet werden, da Standards durch ihren Reifeprozess ein gewisses Maß an Qualität, eine erhöhte Akzeptanz und Interoperabilität der Modelle sowie die größtmögliche Werkzeugunterstützung versprechen.

2.3.2 Anforderungen an die Laufzeitumgebung

Zur Laufzeit wird eine Kompositionsinfrastruktur zur Interpretation des oben motivierten Modells benötigt, die die dynamische Suche und Integration verteilter Komponenten sowie die Ausführung der universellen Komposition umsetzt. Diese Systeminfrastruktur muss sich den folgenden Anforderungen stellen:

Modellverwaltung Sowohl im Autorenprozess als auch zur Laufzeit muss der Zugriff auf die Komponenten-, Kompositions- und Domänenmodelle gesichert sein. Deshalb muss mindestens eine Instanz – vergleichbar mit der *Service Registry* in SOA – für deren Verwaltung existieren. Neben dem Zugriff auf die Modelle sollte sie deren Konsistenz und Validität sicherstellen.

Späte Bindung Im Gegensatz zu konventionellen Diensten, die entfernt in beliebigen Technologien ausgeführt werden können und ein wohldefiniertes Ergebnis liefern, macht die Komposition von bzw. mit UI-Bestandteilen die Ausführung

und Kopplung von Komponenten in der Client-Umgebung des Nutzers nötig. Ausgehend von den Abstraktionsmechanismen des Kompositionsmodells müssen deshalb kompatible Komponenten zur Laufzeit – nach dem Vorbild der „späten Bindung“ von Web Services – integriert werden.

Discovery Die Auswahl und späte Bindung von Komponenten soll in Abhängigkeit vom System-, Nutzer- und Ausführungskontext erfolgen. Dies macht entsprechende *Discovery*-Algorithmen nötig, die bei der Auswahl sowohl *funktionale* Gesichtspunkte als auch *nicht-funktionale* und *Kontexteigenschaften* einbezieht. Die Kandidatenmenge muss entsprechend gefiltert und gewichtet werden, bis die „beste“ Komponente gefunden ist.

Life-Cycle-Management Zur Sicherstellung der Funktionsfähigkeit der kompositen Anwendung zur Laufzeit muss die Plattform eine Reihe grundlegender Funktionen bereitstellen. Dabei ist die Verwaltung des Lebenszyklus' der Komponenten von der Integration, Komposition, Initialisierung und Ausführung bis hin zur korrekten Entfernung eine zentrale Aufgabe.

Lose Kopplung Zur Kommunikation und Koordination innerhalb der Komposition, d. h. zur Realisierung des Kontroll- und Datenflusses, muss die Laufzeitumgebung – unter Beachtung der losen Kopplung aller Bestandteile – entsprechende Mechanismen bereitstellen. In diesem Zusammenhang ist auf Fehlertoleranz zu achten, d. h. beim Ausfall von Diensten muss reagiert und die Komposition angepasst werden, z. B. durch Austausch der betroffenen Komponenten mit funktional äquivalenten Alternativen.

Mediation Im Sinne der angestrebten Interoperabilität muss die Mediation auf der Datenebene ermöglicht werden. Dies beinhaltet sowohl die Überbrückung syntaktischer Differenzen zwischen Modellabstraktion und tatsächlicher Implementierung als auch zwischen verschiedenen, separat entwickelten Komponenten selbst, falls diese unterschiedliche Datenmodelle nutzen.

Kontextualisierung Komposite Anwendungen sollen zur Laufzeit auf Kontextänderungen reagieren können. Neben der korrektiven Adaption zur Aufrechterhaltung der Funktionalität sollten Adaptionen qualitativer Natur unterstützt werden, die vom Entwickler beschrieben werden. Dies impliziert die Konzeption und Realisierung von Mechanismen zur Umsetzung der Kontrollschleife aus Kontexterfassung, -modellierung und -nutzung als Teil der Laufzeitumgebung.

Zustandstransfer Auch hinsichtlich der Adaptionstechniken müssen die Besonderheiten des zustandsbehafteten Komponentenmodells beachtet werden. Eine besondere Rolle kommt deshalb Konzepten des Zustandserhaltes und -transfers zu, die u. a. beim dynamischen Komponententausch Anwendung finden müssen.

Kontextverwaltung Grundlage für den dynamischen Integrationsprozess und die Adaption zur Laufzeit bildet Kontextwissen, welches dynamisch erfasst, modelliert, validiert und ggf. erweitert werden muss. Aus Gründen der plattform- und anwendungsübergreifenden Nutzung ist eine Trennung zwischen adaptiver Laufzeitumgebung und den Kontextverwaltungsmechanismen anzustreben. Hierzu bietet sich eine eigenständige, dienstbasierte Lösung an, die mit den Überwachungs- und Nutzungskonzepten potentiell verschiedener Kompositionsplattformen gekoppelt wird.

Auch im Hinblick auf die Konzepte der Laufzeitumgebung müssen nicht-funktionale Anforderungen wie die **SoC**, die **Erweiterbarkeit** und die **Standardkonformität** Beachtung finden. Bei aller Abstraktion und Vereinfachung sollen auf der anderen Seite die **Stabilität** und **Performanz** der Anwendungen nicht negativ beeinflusst werden.

In diesem Kapitel wurden die wichtigsten Begrifflichkeiten der Arbeit vorgestellt und in den Forschungskontext eingeordnet. Ausgehend von drei Beispielszenarien wurden funktionale und nicht-funktionale Anforderungen an die zu entwickelnden Konzepte formuliert. Das folgende Kapitel widmet sich nun der Analyse und Bewertung bestehender Ansätze aus der Forschung und Technik hinsichtlich der identifizierten Herausforderungen.

3

Entwicklung adaptiver, kompositer Webanwendungen – Stand der Forschung und Technik

Das folgende Kapitel gibt einen Überblick über den Stand der Forschung und Technik hinsichtlich der Entwicklung und Systemstützung adaptiver, kompositer Web- bzw. Mashup-Anwendungen. Dazu erfolgt die Betrachtung und Bewertung verwandter Ansätze aus den verschiedenen, durch die Zielstellung tangierten Forschungsgebieten im Hinblick auf die im vorigen Kapitel skizzierten Heraus- und Anforderungen.
Die Aufteilung des Kapitels folgt den beiden wesentlichen Stoßrichtungen aus Forschung und Technik, um den motivierten Herausforderungen zu begegnen:

SOA Es erfolgt die Untersuchung von Konzepten zur Komposition von Anwendungen aus verteilten Diensten im Sinne von SOA unter Berücksichtigung der identifizierten Herausforderungen. Dazu zählen die plattformunabhängige Beschreibung, die kontextsensitive Auswahl und Integration von Anwendungsbestandteilen, die Auflösung von Interoperabilitätsproblemen und nicht zuletzt die Unterstützung von Nutzerinteraktionen in Dienstkompositionen.

Web Engineering Hier werden Ansätze zur strukturierten Entwicklung von Webanwendungen und Mashups vorgestellt, die die genannten Herausforderungen unterstützen und Paradigmen des SOC, wie die Verteilung und späte Bindung von Diensten, für interaktive Anwendungen beachten.

Die gewonnenen Erkenntnisse dienen als Grundlage für die Konzeption der Modellierung (Kapitel 5) und Kompositionsinfrastruktur (Kapitel 6).

3.1 SOA und Dienstkomposition zur Interaktion mit Diensten

Wie in Abschnitt 2.1.1 beschrieben, basiert die Idee der kompositen Anwendungen und Mashups auf den Konzepten des Service Oriented Computing (SOC). Die Standards, Frameworks und Forschungsansätze aus diesem Gebiet gehen i. d. R. jedoch von vollautomatischen Kompositionen aus und bieten keine Konzepte für die Interaktion mit Nutzern oder die Bereitstellung von UIs. Der folgende Abschnitt wird zeigen, dass die dafür entwickelten Konzepte und Modelle in vielerlei Hinsicht dennoch als Vorbild für universelle Kompositionen dienen können.

Nach Alonso et al. (2004) tragen zur Service-Komposition verschiedene Modelle bei: Das *Komponentenmodell* repräsentiert die unterstützten Dienste, das *Orchestrierungsmodell* regelt die Reihenfolge und Bedingungen der Dienstaufrufe, *Datenmodelle* beschreiben die zugrunde liegenden Datenstrukturen und den Datenfluss, und *Dienstauswahlmodelle* kapseln die Logik hinter der späten Bindung von Diensten, die bei (semi-)automatischen Kompositionsansätzen eine tragende Rolle spielt. All jene Modelle sind auch im Kontext dieser Arbeit relevant – sie müssen jedoch unter dem Gesichtspunkt der universellen Komposition untersucht werden.

Die Verfahren zur Service-Komposition können nach Dustdar und Schreiner (2005) entlang zweier Dimensionen kategorisiert werden, wobei zwischen *statischen* und *dynamischen* sowie zwischen *manuellen* und *automatischen* Ansätzen unterschieden wird. Abbildung 3.1 zeigt anhand des Spannungsfeldes aus Zeitpunkt der Dienstbindung und Art der Planung die verschiedenen Kompositionsstrategien. Während in statischen, vorhersehbaren Anwendungen und Prozessen die manuelle Erstellung des Kompositionsplans mit vordefinierten Diensten vorherrscht, bedingen dynamische, veränderliche Umgebungen die späte Bindung (*Late Binding*) und ggf. dynamische Erstellung des Kompositionsplans in Abhängigkeit des vorherrschenden Kontextes.

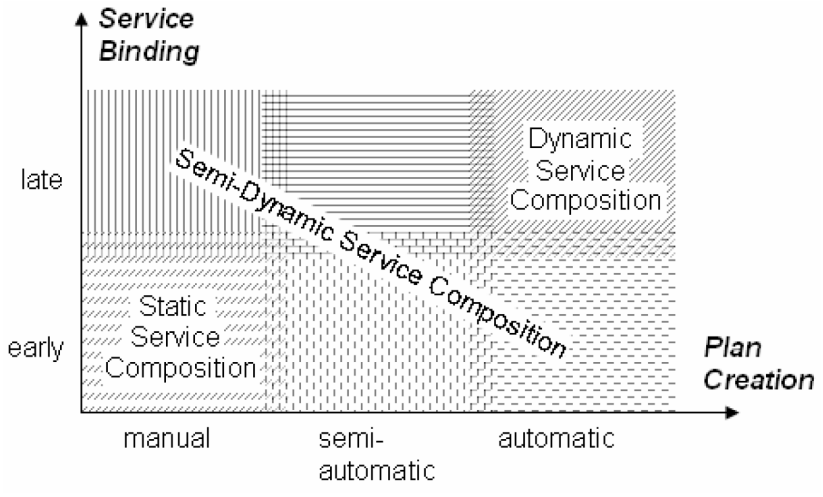

Abb. 3.1: Klassifikation von Strategien zur Service-Komposition [Fluegge et al., 2006]

Beide Herangehensweise liefern für die vorliegende Arbeit konzeptionelle Anhaltspunkte. Die statischen Vertreter versprechen Konzepte zur Modellierung von Diensten und verteilten, kompositen Anwendungen, die u. U. auf interaktive Mashups übertragen werden können. Vor dem Hintergrund der angestrebten späten Bindung in

universellen Kompositionen sind zudem die Konzepte der dynamischen Vertretet der Web-Service-Komposition von hoher Relevanz. Sie werden im Anschluss diskutiert. Zuletzt werden dedizierte Lösungen zur Unterstützung von Adaptivität und der Interaktion mit Diensten und Dienstkompositionen vorgestellt und hinsichtlich der im letzten Kapitel gesammelten Anforderungen bewertet.

3.1.1 Statische Dienstkomposition

Traditionelle Dienstkompositionen verfolgen einen statischen, manuellen Entwicklungsansatz zur Abbildung von Geschäftsprozessen auf eine Kette von Dienstaufrufen. Bei den Diensten handelt es sich um Web Services, die mittels WSDL [@Chinnici et al., 2007] beschrieben sind und über SOAP [@Gudgin et al., 2007] kommunizieren. Der Daten- und Kontrollfluss zwischen ihnen wird i. d. R. mittels Business Process Execution Language (BPEL) [@Alves et al., 2007] beschrieben – einer deklarativen Beschreibungssprache, welche durch entsprechende *Engines* interpretiert wird und zur vollautomatischen, prozessorientierten Ausführung der Kompositionen führt.

Dustdar und Schreiner (2005) geben einen guten Überblick über bestehende Konzepte zur statischen Service-Komposition. Deren Modelle und Systeme eignen sich jedoch nur bedingt für Mashups im Sinne der Arbeit, da sie schwerpunktmäßig die Verknüpfung von Diensten im Hinblick auf ihr Verhalten (*behavioral aggregation*) adressieren. Bei Mashups steht hingegen nicht die Beschreibung von Prozessen sondern die Integration von Daten und Benutzerschnittstellen im Vordergrund [Curbera et al., 2007; Beemer und Gregg, 2009], weshalb datenflussorientierte Modelle geeigneter als Workflow-orientierte Ansätze sind [Abiteboul et al., 2008].

Aus Sicht der IT-Dienstleister besteht zunehmend die Notwendigkeit, menschliche Aktivitäten in Geschäftsprozessen beschreiben zu können. Aufbauend auf ein White Paper von IBM und SAP aus dem Jahr 2005 veröffentlichte deshalb ein Industriekonsortium aus namhaften Softwareherstellern (u. a. Adobe, IBM, Oracle und SAP) im Jahr 2007 zwei Spezifikationen, die die Integration menschlicher Aktivitäten in BPEL erlauben und im Augenblick die Standardisierung durch die Organization for the Advancement of Structured Information Standards (OASIS) erfahren[1]. Die wichtigste Erweiterung von BPEL4People (B4P) [@Agrawal et al., 2007] gegenüber BPEL ist die Einführung der *PeopleActivity*, welche eine menschliche Aktivität als Teilschritt eines Geschäftsprozesses repräsentiert – analog zum Aufruf eines Web Services. Die Definition der eigentlichen Inhalte der Aktivität erfolgt in Form einer WS-HumanTask-Beschreibung (WS-HT) [@Clément et al., 2010], welche in die Aktivität eingebettet oder daraus referenziert wird.

Human Tasks (HT) können als Dienste verstanden werden, die von Personen ausgeführt werden und sowohl über eine Schnittstelle zum Prozess als auch über eine Schnittstelle zum Nutzer verfügen. Sie könnten also als UI-Komponenten aufgefasst werden. Die WS-HT-Spezifikation legt den Fokus der Modellierung aber ausschließlich auf die Definition von Rollen sowie die Interaktion mit dem Prozess über Ein- und Ausgaben des HT. Die Darstellung der Tasks zur Interaktion mit dem Nutzer übernehmen *Task List Clients*, wobei die Art und Form der Präsentation herstellerspezifisch erfolgt. Weder die Definition der Benutzerschnittstelle noch die

[1] OASIS BPEL4People Working Group: http://www.oasis-open.org/committees/bpel4people/

Komposition mehrerer Tasks in einer Oberfläche können mit WS-HT definiert werden. Auch für die Anwendung des *Late-Binding*-Prinzips auf HT gibt es keine Konzepte.

Als „leichtgewichtige“ Alternativen zu BPEL wurden verschiedene DSLs zur Spezifikation Workflow-basierter Anwendungen vorgeschlagen. So bieten Bite [Curbera et al., 2007] und JOpera [Pautasso, 2009] eigene Choreographiesprachen, welche Anwendungsbestandteile auf Basis eines Representational State Transfer (REST)-basierten Modells miteinander verknüpfen. Verschiedenste Ressourcen in Form von Java-Anwendungen, SOAP und REST Services können auf dieses universelle Modell abgebildet werden. Bei der Unterstützung menschlicher Interaktionen unterliegen die Ansätze jedoch den gleichen Beschränkungen wie BPEL.

Neben der *Orchestrierung* durch prozessorientierte Sprachen kann auch ein *struktureller* Ansatz wie in Service Component Architecture (SCA) verfolgt werden. Dieser geht von expliziten Abhängigkeiten (*dependencies*) aus, d. h. jede Komponente eines *Moduls* stellt Dienste bereit und kann wiederum von Diensten anderer Komponenten (auch aus anderen Modulen) abhängig sein. Während im prozessorientierten Modell die Reihenfolge von Dienst- bzw. Operationsaufrufen explizit definiert wird, ergibt sie sich im strukturellen Modell durch die Geschäftslogik der Komponenten selbst und das SCA-System ist für die Auflösung dieser Abhängigkeiten zuständig. Beide Ansätze schließen sich deshalb nicht notwendigerweise aus, sondern können als komplementär betrachtet werden: BPEL kann beispielsweise zur Spezifikation der Abläufe innerhalb einer Komponente der SCA dienen.

In jedem Fall scheint das Modell der SCA besser für die Spezifikation komponentenbasierter Mashup-Anwendungen geeignet zu sein, da es sich stark an den Strukturen klassischer Softwaresysteme orientiert. Gerade bei interaktiven Anwendungen lassen sich Abläufe nicht mehr ohne weiteres als Prozess formalisieren, da vielfältige Nutzerinteraktionen die Komplexität erhöhen. Allerdings existieren für SCA keinerlei Konzepte zur Integration menschlicher Aktivitäten oder gar zur Komposition von Benutzerschnittstellen. Zudem ist die Existenz expliziter Abhängigkeiten zwischen Komponenten problematisch, da sie der losen Kopplung widerspricht.

Fazit

Zusammenfassend ist zu sagen, dass sich die Modelle der statischen Dienstkomposition – sowohl prozessorientiert als auch strukturell – im Wesentlichen durch ihre Technologie- und Plattformunabhängigkeit auszeichnen. Sie erlauben die Verknüpfung potentiell verteilter Anwendungsbestandteile über ihre Schnittstellen, beschränken sich hierbei allerdings auf Komponenten der Daten- und Geschäftslogikebene. Somit werden lediglich funktionale Abhängigkeiten sowie Daten- und Kontrollfluss innerhalb der Kompositionen beschrieben. Für die Anforderungen der Benutzerschnittstelle, wie die Definition von Layouts, Sichten und Nutzerinteraktionen, oder die Unterstützung zustandsbasierter Ressourcen, bedarf es Erweiterungen oder gänzlich neuer Konzepte.

3.1.2 Dynamische Dienstauswahl und -Komposition

Ansätze zur **dynamischen Dienstauswahl** gehen einen Schritt weiter und nutzen die Trennung von Schnittstelle und Implementierung bei Diensten aus, um die Bindung letzterer möglichst spät und kontextabhängig durchzuführen. Abbildung 3.2

illustriert die verschiedenen Möglichkeiten am Beispiel von BPEL. Zur Entwicklungszeit müssen die Dienstarten bekannt sein, beim Deployment zumindest die konkreten Dienstschnittstellen, und spätestens zum Zeitpunkt des Aufrufs muss feststehen, unter welchem Endpunkt ein Dienst erreichbar ist. Dadurch ergeben sich große Spielräume beim Zeitpunkt der Bindung. Diese kann dabei bereits bei der Komposition, bei der Kompilierung, beim Deployment, beim Start der Anwendung oder erst kurz vor dem Aufruf erfolgen. Zusätzlich kann die Bindung zur Laufzeit nötig werden, um Fehlern und Dienstausfällen zu begegnen [Pautasso und Alonso, 2005]. Verschiedene wissenschaftliche Arbeiten versuchen, die Dienstbindung entsprechend spät durchzuführen, um größtmögliche Flexibilität und Kontextadaptivität bei der Auswahl zu ermöglichen.

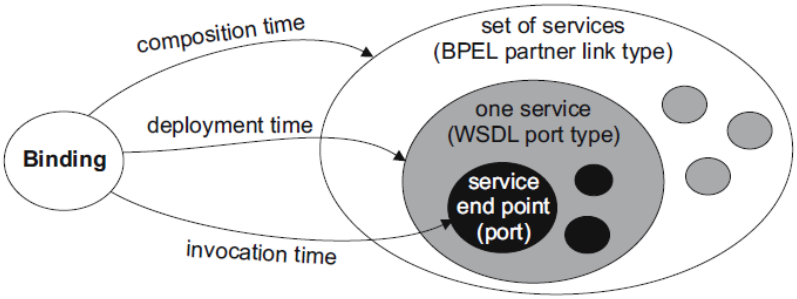

Abb. 3.2: Alternativen der Dienst-Bindung [Pautasso und Alonso, 2005]

Das *WS Binder* Framework von Penta et al. (2006) ist ein typischer Vertreter der konventionellen Dienstkomposition, welcher sich der dynamischen Bindung in Abhängigkeit von funktionalen und nicht-funktionalen Eigenschaften widmet. Zur Beeinflussung der Bindung können globale QoS-Ziele mit dem Ziel der Maxi- oder Minimierung, QoS-Beschränkungen, lokale Beschränkungen für Dienste, z. B. bezüglich Kosten oder Antwortzeit, sowie Abhängigkeiten und explizite White- oder Blacklists formuliert werden. Wie bei der Mehrzahl derartiger Systeme erfolgt die Überwachung der Kriterien auch zur Laufzeit, was im Fehlerfall zum Austausch des betroffenen Dienstes führt. Auf derartige und weitere Mechanismen zur Laufzeitadaptivität wird später in Abschnitt 3.1.3 näher eingegangen. Zur Optimierung des Verfahrens und zur Vergrößerung der Treffermenge bei der Suche passender Dienste kommen zunehmend semantische Schnittstellenbeschreibungen zum Einsatz. Zur Formulierung von *Templates* in existierenden Prozessen nutzen Küster und König-Ries (2007) beispielsweise die DIANE Service Description (DSD). Diese semantische Beschreibung wird zur Laufzeit an eine entsprechende Middleware weitergeleitet, die die vollautomatische Dienstsuche und -Integration in den Prozess übernimmt.
Auch jenseits von BPEL existieren Konzepte für derartige „Platzhalter“: Blau et al. (2009) wenden das Konzept im reMash!-System auf RESTful Mashups an. Dazu werden sog. *blueprints* definiert – Abstraktionen, deren Instanziierungen konkrete Mashups darstellen und denen die DSL Bite [Curbera et al., 2007] als Formalismus zur Beschreibung des Datenflusses dient. Abbildung 3.3 zeigt einen solchen Blueprint mit drei abstrakten Komponenten, für die jeweils einige Alternativen definiert sind. Die Bindung jeweils einer Web-Ressource erfolgt zur Laufzeit, je nach Verfügbarkeit und definierter Anforderungen. Grundlage hierfür bietet die semantische Auszeichnung der beinhalteten Dienste, auf deren Basis ein

Reasoner mögliche Alternativen berechnet und valide Instanziierungen vorschlägt. Dazu greift er auf *Policies* zurück, die QoS-Anforderungen über Regeln der Semantic Web Rule Language (SWRL) ausdrücken. Sind alle Ressourcen gebunden, kann der Plan in einen ausführbaren Prozess transformiert werden.

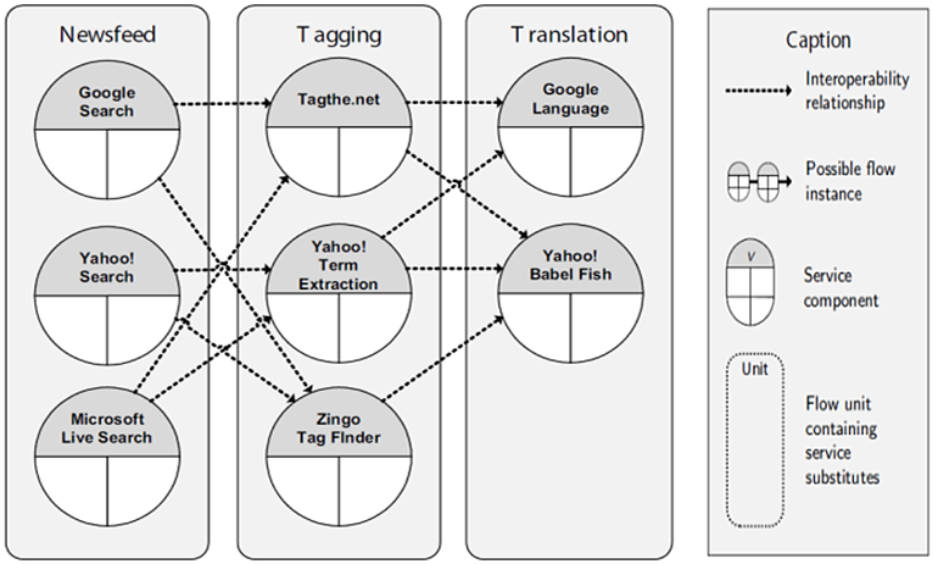

Abb. 3.3: Komposition mit Dienstalternativen in reMash! [Blau et al., 2009]

An der Abbildung ist zu erkennen, dass der Ansatz, wie auch WS Binder zuvor, von einem sequentiellen Datenfluss ausgeht und in einem ausführbaren Prozess endet. Wie bereits im letzten Abschnitt festgestellt wurde, ist die prozessorientierte Sicht jedoch nicht zur Modellierung interaktiver Anwendungen geeignet, da sie den nutzer- bzw. ereignisgetriebenen Daten- und Kontrollfluss nicht geeignet abbildet. Interaktion kann darin lediglich „innerhalb“ von Ressourcen stattfinden – UI-Komposition samt der Definition von Layout-Informationen ist nicht vorgesehen. Im Fall von reMash! ist das Auswahlkonzept zudem auf festgelegte Alternativen beschränkt.

Neben der dynamischen Auswahl und Bindung widmet sich die Forschung der **dynamischen Komposition von Diensten**, die i. d. R. als zustandsbasiertes Planungsproblem aufgefasst wird. Der Nutzer tätigt – teilweise in natürlicher Sprache – eine Eingabe, woraufhin das System auf Basis inverser Transformationen mögliche Ziele ermittelt und für diese Dienstkompositionen berechnet. Dazu bedient es ich formaler Grundlagen aus dem Bereich der künstlichen Intelligenz (*AI Planning*) und ermittelt ausgehend von einem initialen Zustand und einer Menge durchführbarer Aktionen bzw. verfügbarer Dienste mit Vor- und Nachbedingungen Kompositionen, die zu einem Zielzustand führen. Zuletzt wählt der Nutzer das gewünschte Ziel bzw. eine spezifische Komposition aus, die ausgeführt werden soll. Der Ausführungsplan bzw. das Kompositionsmodell ergibt sich somit im Gegensatz zu den bislang vorgestellten Ansätzen erst zur Laufzeit. Nebenbei kann die Planung auch rückwärts verkettet stattfinden, also vom Zielzustand aus beginnen.
Ein typischer Vertreter dynamischer Dienstkompositionen ist CoSMoS (*Component Service Model with Semantics*) [Fujii und Suda, 2009]. Hier werden Dienste durch einen semantischen Graphen repräsentiert, welcher funktionale Semantik, Datentyp- und Verhaltenssemantik beschreibt. Abbildung 3.4 stellt den Ablauf der Komposition

mit den beteiligten Agenten dar: Ausgehend von einer natürlichsprachigen Eingabe wird eine semantische Nutzeranfrage generiert, auf deren Basis durch den *Service Composer* ein Ausführungspfad berechnet wird. Durch semantisches Matching sucht ein *Reasoner* Dienstimplementierungen entlang des Ausführungspfades, die die erforderlichen Ein- und Ausgaben, Funktionalitäten und Kontexteigenschaften aufweisen. Zuletzt wird der Pfad durchlaufen, d. h. die gefundenen Dienste werden gebunden.

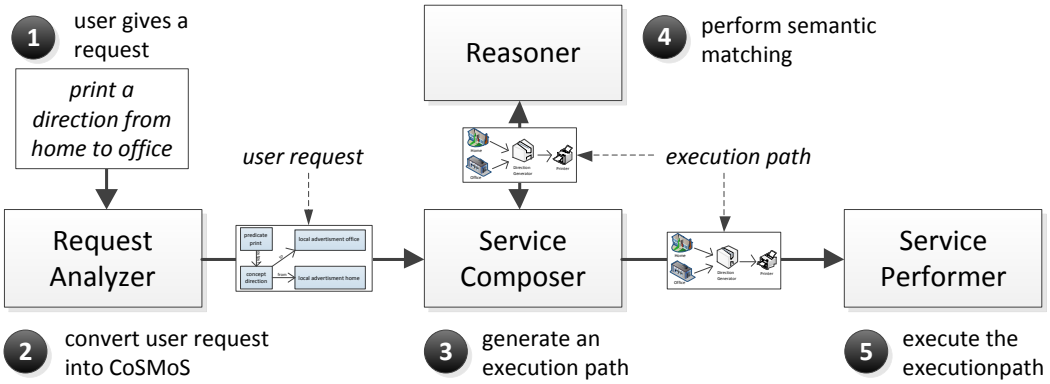

Abb. 3.4: Ablauf der Komposition im SeGSeC-System [Fujii und Suda, 2004]

Auch wenn die dynamische Komposition nicht im Fokus dieser Arbeit steht, so sind die dabei genutzten Verfahren im Hinblick auf die Konzeption höchst relevant. Sie basieren auf semantischen Dienst- und Zielbeschreibungen, weshalb man von *Semantic Web Services (SWS)* spricht. Der Prozess der Suche, Auswahl, Integration und Komposition von Diensten auf Basis funktionaler und nicht-funktionaler semantischer Beschreibungen – auch als SWS Nutzungsprozess bezeichnet (vgl. Abbildung 3.5) – kann als Referenz für die universelle semi-automatische Komposition von Anwendungen, ausgehend von einer plattformunabhängigen semantischen Beschreibung, dienen. Im Folgenden wird dieser Prozess überblicksartig vorgestellt – zur Vertiefung sei auf die Darstellung von Arroyo et al. (2004) verwiesen.

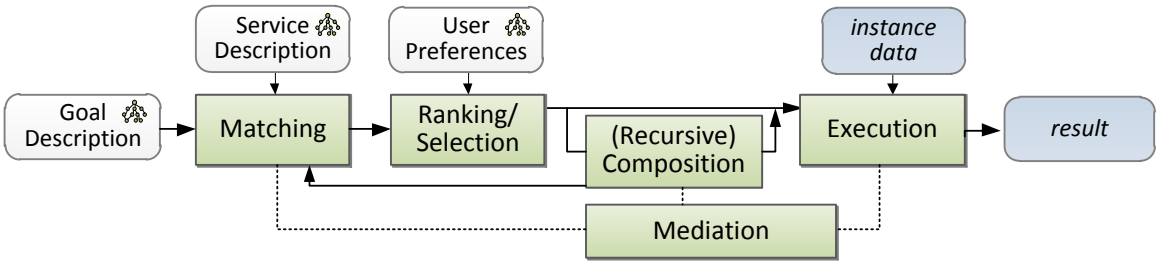

Abb. 3.5: Phasen im SWS-Nutzungsprozess

Dienst- und Zielbeschreibung

Grundlage jeder semantischen Komposition sind semantische Dienstbeschreibungen, die zugleich Ausgangspunkt für zu planende Kompositionen und abstrakte Repräsentationen zu bindender Dienste sein können. Hinsichtlich der Beschreibungsformen wird zwischen *bottom-up*- und *top-down*-Ansätzen unterschieden. Letztere, wie z. B. OWL-S [Martin et al., 2007b] und WSMO [Fensel et al., 2008], modellieren

Dienstsemantik möglichst vollständig, während technische und Implementierungsdetails durch *Grounding*-Mechanismen verdeckt werden. Diese Ansätze sind i. d. R. aufgrund der gewählten Sprachmittel sehr ausdrucksstark, allerdings auf Kosten der Handhabbarkeit bei der Erstellung und Performanz bei der Verarbeitung.

Dienstbeschreibungen nach dem *bottom-up*-Prinzip gehen den umgekehrten Weg: Sie erweitern existierende Beschreibungsformate mit semantischen Annotationen bzw. Referenzen zu Konzepten separater Ontologien. Semantic Annotations for WSDL and XML (SAWSDL) [@Farrell und Lausen, 2007] stellt beispielsweise das Attribut *modelReference* bereit, um Teile einer WSDL-Schnittstelle mit semantischen Konzepten zu verknüpfen. Durch die Definition von *schema mappings* in Form von *Liftings* (in semantische Konzepte) und *Lowerings* in die entgegengesetzte Richtung werden semantisch verwalteten Informationen auf die Protokolle und Technologien der syntaktischen Ebene abgebildet. Im Fall semantisch typisierter Daten eines Web Services erfolgt beispielsweise die Abbildung von Ontologiekonzepten auf eXtensible Markup Language (XML)-Instanzen gemäß den in der WSDL definierten XML-Schemata. Leider ist SAWSDL ohne Konkretisierungen der Annotationsformen und -ziele nur eingeschränkt nutzbar [Martin et al., 2007a]. Deshalb wurden mit WSMO-Lite [Kopecký und Vitvar, 2008] weitere Konzepte eingeführt, u. a. spezifische Annotationstypen zur Definition von funktionaler und Verhaltenssemantik.

Abbildung 3.6 veranschaulicht das Konzept: Eine initiale Dienstbeschreibung ① wird so erweitert, dass die Schnittstellenbestandteile (Operationen, Ein- und Ausgaben) über Annotationen mit Domänenkonzepten ② (rot) verknüpft werden können. Die semantische Beschreibung wird dann publiziert ③ und kann zur Suche nach kompatiblen Diensten für eine Anfrage zur Laufzeit herangezogen werden ④. Die Anfrage bzw. das *Template* ⑤ ist ebenso semantisch typisiert, was einen direkten Abgleich von benötigter und verfügbarer Schnittstelle erlaubt. Der Vorteil der Nutzung semantischer Modelle liegt in der Vererbungshierarchie im Modell: Im Gegensatz zum syntaktischen Vergleich können auch „unscharfe“ Treffer unterstützt werden, wie im Beispiel anhand des *Inputs* gezeigt. Template- und Dienst-Eingabe referenzieren verschiedene Konzepte, die jedoch im semantischen Modell durch eine Subklassenbeziehung verbunden sind. Diese Relation kann aufgelöst werden, d. h. es wird zwischen den Schnittstellen mediiert.

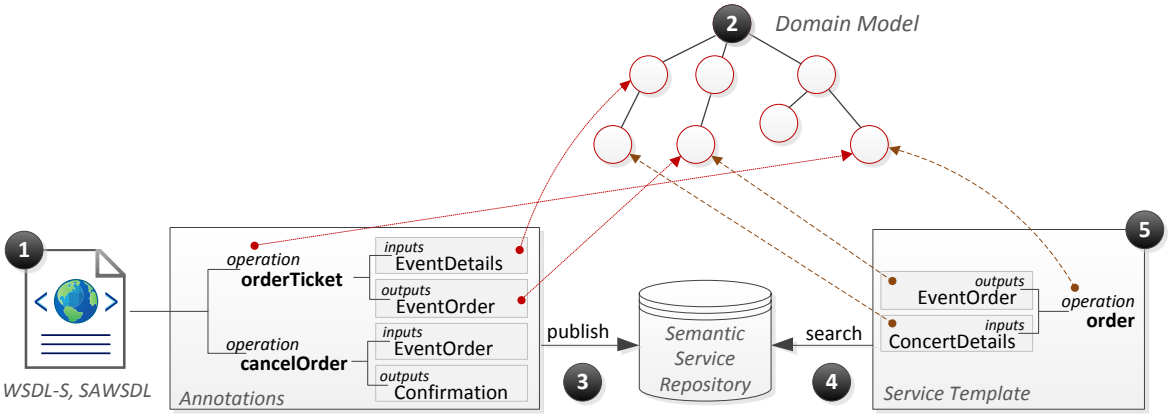

Abb. 3.6: Matchmaking semantisch annotierter Dienste am Beispiel von SAWSDL (nach [Sivashanmugam et al., 2003])

Ähnliche Modelle existieren für REST-basierte Dienste. Im Ansatz von Sheth et al. (2007) zur Suche und Integration von RESTful Services wird beispielsweise auf RDFa- oder GRDDL-Annotationen zurückgegriffen, die in die Web-Ressourcen eingebettet sind. Das System extrahiert diese semantischen Beschreibungen zur Laufzeit und ermöglicht dem Nutzer, die entsprechenden Dienste grafisch zu verbinden. Als Möglichkeit zur Abstraktion von Diensten in vorgefertigten BPEL-Prozessen wurde bereits die DIANE Service Description genannt [Küster und König-Ries, 2007], doch es besteht keinerlei Beschränkung auf klassische Web Services: Yao und Etzkorn (2004) zeigen, dass sich WSDL/RDF-Definitionen gleichsam zur Beschreibung von Software-Komponenten nutzen lassen.

Die Anfrage an das System zur dynamischen Komposition erfolgt i. d. R. in Form einer der vorgestellten semantischen Dienstbeschreibungen. Einige Sprachen, wie Web Service Modeling Ontology (WSMO), unterscheiden zwischen Konzepten des Dienstangebots und -ziels, während andere, wie OWL-S, für beide das gleiche Vokabular nutzen. In der Mehrzahl der Forschungsansätze wird versucht, die Anfragen weiter zu vereinfachen, z. B. indem die Eingabe natürlichsprachiger Nutzeranfragen analysiert und automatisch in entsprechende Beschreibungen transformiert wird (vgl. [Yao und Etzkorn, 2004; Fujii und Suda, 2009]).
Beide Arten semantischer Dienstmodelle können als Grundlage zur Beschreibung konkreter und abstrakter Anwendungsbausteine i. S. v. Mashups dienen. Dafür sind jedoch Erweiterungen nötig, um u. a. zustandsgebende Eigenschaften abbilden zu können, wie sie für interaktive Komponenten benötigt werden.
Darüber hinaus besteht die Beschränkung der Modelle weniger in der Ausdrucksstärke, als vielmehr in den verwendeten Referenzierungsarten und verwendeten Domänenkonzepten. Deren Semantik bezieht sich auf Datentypen und Funktionalitäten, aber naturgemäß nicht auf Konzepte, die die Interaktion mit Nutzern widerspiegeln. Die Annotation mit Interaktionssemantik (Interaktionstechniken, Modalitäten oder unterstützte menschliche Aktionen im Sinne von Task-Modellen) und QoS-Kriterien hinsichtlich der Präsentation und Interaktion ist mit bestehenden Ansätzen somit nicht möglich.

Matchmaking

Der erste Schritt des SWS-Nutzungsprozesses aus Abbildung 3.5 widmet sich der **Auffindung** von Dienstkandidaten, die den Anforderungen hinsichtlich der semantisch formalisierten Zielbeschreibung (*Goal*) genügen. Der Abgleich zwischen Angebot und Nachfrage wird als *Matching* oder *Matchmaking* bezeichnet.
Die dazu eingesetzten Methoden sind entweder logikbasiert, nicht-logikbasiert oder hybrid [Klusch, 2008]. Logikbasierte Ansätze, wie [Chabeb et al., 2010], ermitteln den Übereinstimmungsgrad zwischen Ziel- und Dienstschnittstelle über die logischen Beziehungen zwischen den annotierten Konzepten. Nicht-logikbasierte Ansätze nutzen Ähnlichkeitsmaße auf Text- und Graph-Ebene, z. B. die textuelle Übereinstimmung von Operationsnamen, ggf. unter Einbeziehung von Thesauri und Wörterbüchern. Durch die ihnen innewohnende Ungewissheit werden sie nur zur Entwicklungszeit eingesetzt, um die semi-automatische Komposition zu unterstützen. Hybride kombinieren beide Verfahren und versprechen eine höhere Präzision bei der Auffindung, leiden aber gleichsam unter den Problemen der nicht-logikbasierten Ansätze, z. B. beim Einsatz unterschiedlicher Sprachen.

Grundgedanke beim Matching auf Basis von Beschreibungslogik ist die Auffassung von *Request* (R) und *Advertisement* (A) als Ontologiekonzepte, aus denen der Übereinstimmungsgrad berechnet wird. Dazu werden Subsumptionsbeziehungen im semantischen Modell ausgewertet. Wenn C von D subsumiert wird ($C \sqsubseteq D$), dann ist die Menge der Instanzen von C ein Teilmenge der Instanzen von D. Daraus ergeben sich nach L. Li und Horrocks (2003) die folgenden Grade der Übereinstimmung in absteigender Ordnung:

Exact $R \equiv A$: R und A sind äquivalent.

PlugIn $R \sqsubseteq A$: R wird von A subsumiert, d. h. R ist Subkonzept von A.

Subsume $A \sqsubseteq R$: A wird von R subsumiert, d. h. R ist ein Superkonzept von A.

Intersection $\neg(R \sqcap A \sqsubseteq \bot)$: Es existieren Überschneidungen zwischen R und A.

Disjoint $R \sqcap A \sqsubseteq \bot$: Es existieren keinerlei Überschneidungen zwischen R und A.

Die Auswertung der Beziehungen richtet sich u. a. nach der *Intention* von Dienstnutzer und -anbieter, welche nach Fensel et al. (2008) *universell* (alle Konzepte gefordert) oder *existentiell* (einige Konzepte gefordert) sein kann. Eine mengentheoretische *Intersection* entspricht folglich einem *Match*, falls das Angebot universell und die Anfrage existentiell verstanden werden, jedoch nicht anders herum.

Entsprechende Matchmaking-Verfahren finden in einer Vielzahl von Forschungsarbeiten, u. a. von Jaeger et al. (2005); Hristoskova et al. (2010) und Chabeb et al. (2010), Anwendung. Da das angestrebte Konzept die automatische Auffindung von Komponenten zur Laufzeit unterstützen soll, muss die Passgenauigkeit von Komponenten gewährleistet sein, weshalb folglich nur logikbasierte Algorithmen als Ausgangspunkt dienen können.

Jaeger et al. (2005) beschreiben einen typischen Vertreter, der neben Ein- und Ausgaben der Dienste zusätzlich funktionale Kategorien und nutzerspezifische Anforderungen berücksichtigt, die allesamt mit OWL-S formalisiert sind. Zunächst werden die semantischen Konzepte der Ein- und Ausgaben sowie der Kategorie nach dem beschriebenen Verfahren abgeglichen, wobei nur die Übereinstimmungsgrade *Exact* und *Subsumes* unterstützt werden und der Vergleich ansonsten fehl schlägt. Wichtig im Hinblick auf das spätere Konzept ist der Aspekt der *Contravariance*: Die Subsumes-Beziehung zwischen Anfrage und Angebot ist für Ein- und Ausgaben genau umgekehrt definiert. Es gelten $A_{input_i} \sqsubseteq R_{input_j}$ sowie $R_{output_n} \sqsubseteq A_{output_m}$. Dadurch wird sichergestellt, dass entlang des Datenflusses immer nur *Up-Casts* stattfinden und niemals mehr Informationen benötigt werden, als angeboten werden können (vgl. Erläuterung am Beispiel in Abschnitt 6.1.2.1). Im nächsten Schritt werden alle verbliebenen Kandidaten hinsichtlich nutzerdefinierter Anforderungen geprüft. Diese umfassen Aspekte, die über das Reasoning mittels OWL-S hinausgehen, z. B. die geographische Verfügbarkeit oder die Einhaltung von QoS-Kriterien.

Bei entsprechend großen Dienstmengen stellen solche Vergleiche zur Laufzeit einen immensen Aufwand dar. Im System YASA-M [Chabeb et al., 2010], welches logikbasiertes Matchmaking auf Basis einer SAWSDL-Erweiterung durchführt, wird deshalb die Kandidatenmenge bereits vor der semantischen Überprüfung durch Vergleich der Anzahl von Interfaces, Operationen sowie Ein- und Ausgaben eingeschränkt. Ähnliche Verfahren können auch im späteren Konzept zum Einsatz kommen, um die Performanz zu erhöhen.

Das Matchmaking auf Basis von Subsumptionsbeziehungen zwischen Ontologiekonzepten stellt für das spätere Konzept einen guten Ausgangspunkt dar. Bestehende Ansätze zeigen, dass dadurch die dynamische Auffindung von Diensten unabhängig von syntaktischen Differenzen und mit vertretbarem Aufwand, d. h. ohne Beeinträchtigung der Benutzbarkeit, möglich ist. Ihre Algorithmen können mit Anpassungen an die erweiterten semantischen Beschreibungen auch für die Suche nach Mashup-Komponenten eingesetzt werden.
Eine Beschränkung auf die semantische Ebene ist jedoch unzureichend, da bei der Komposition von Web-Ressourcen insbesondere syntaktische Informationen, wie die verwendeten Datentypen bzw. -schemata, berücksichtigt werden müssen. Dies macht *Schema-Matching*-Verfahren notwendig, die von Shvaiko und Euzenat (2005) ausführlich vorgestellt und klassifiziert werden. Eine vollautomatische Abbildung scheint vor dem Hintergrund der aktuellen Forschung eher fraglich und wenig praktikabel. Ein vielversprechender Ansatz ist jedoch die Verwendung standardisierter *Groundings* der Ontologiekonzepte, auf die von Seiten der Dienst- und Komponentenentwickler eine manuelle Abbildung existieren kann. In diesem Fall kann, wie bei SAWSDL, das syntaktische Matching entfallen.

Durch die Kombination von semantischer und syntaktischer Suche bei hybriden Verfahren kann die Präzision des Matchmakings gesteigert werden. Einen guten Überblick entsprechender Ansätze bieten Klusch (2008). Im Rahmen einer Diplomarbeit [Radeck, 2011] wurden diese und weitere Konzepte ausführlich diskutiert und im Hinblick auf ihre Eignung für das Konzept untersucht.

Ranking

Der nächste Schritt widmet sich der **Rangefolgebildung** (*Ranking*) und Selektion von Kandidaten, basierend auf ihren funktionalen und nicht-funktionalen Eigenschaften. Hierfür existieren einige Konzepte, die die Einbeziehung verschiedener Kriterien basierend auf semantischen Beschreibungen nicht-funktionaler Diensteigenschaften ermöglichen [Fensel et al., 2008] und auch Kontextinformationen in den Auswahlprozess einbeziehen [Gilles et al., 2009; Treiber et al., 2010].
Das von Fensel et al. (2008) vorgestellte Verfahren erlaubt die Rangfolgebildung anhand mehrerer vordefinierter Kriterien. Dazu werden nicht-funktionale Eigenschaften, wie Orts- und Zeitbezug oder QoS-Zusicherungen (Verfügbarkeit, Zuverlässigkeit, Sicherheit), durch Ontologien semantisch modelliert. Auf die Konzepte wird sowohl aus der Dienst- als auch aus der Zielbeschreibung Bezug genommen. Codebeispiel 3.1 zeigt anhand einer exemplarischen Anfrage, wie die Kriterien, d. h. die Non Functional Properties (NFP) in Zeile 4, der Ordnungssinn zur Sortierung (Zeile 5) und die Anzahl auszuwählender Dienste (Zeile 6) in der Web Service Modeling Language (WSML) angegeben werden.

```
1 namespace { _"Goal.wsml#", so _"Shipment.wsml#", up _"UpperOnto.wsml#" ,...}
2 goal Goal1
3   annotations
4     up#nfp hasValue obl#Obligation
5     up#order hasValue pref#ascending
6     up#top hasValue "1"
7   endAnnotations ...
```

Lst. 3.1: Goal mit Angabe einer nicht-funktionalen Eigenschaft in WSML (nach Fensel et al. (2008))

Die referenzierten Kriterien beziehen sich auf Eigenschaften von Diensten, die durch logische Regeln formuliert werden können. Codebeispiel 3.2 zeigt die Definition der oben genutzten NFP *Obligation* (Verbindlichkeit) beim Paketversand: Die Regeln in den Zeilen 7–9 und 11–13 sagen aus, dass die Haftung des Versandunternehmens (*Runners*) für ein beschädigtes oder verlorenes Paket dem deklarierten Wert entspricht (Zeile 11–13), jedoch höchstens 150 $ (Zeile 7–9).

```
1  namespace { _"WSRunner.wsml#", runner _"WSRunner.wsml#", ...}
2  webService runnerService
3    nonFunctionalProperty obligations
4      definition definedBy
5        // in case the package is lost or damaged Runners liability is
6        // the declared value of the package but no more than 150 USD
7        hasPackageLiability(?package, 150):- ?package[so\#packageStatus hasValue ?
             status] and
8          (?status = so\#packageDamaged or ?status = so\#packageLost) and
9          packageDeclaredValue(?package, ?value) and ?value>150.
10
11       hasPackageLiability(?package, ?value):- ?package[so\#packageStatus hasValue ?
             status] and
12         (?status = so\#packageDamaged or ?status = so\#packageLost) and
13         packageDeclaredValue(?package, ?value) and ?value =< 150.
14       ...
15       //in case the package is not lost or damaged Runners liability is 0
16       hasPackageLiability(?package, 0):- ...
17       packageDeclaredValue(?package, ?value):- ...
18   capability runnerOrderSystemCapability
19   interface runnerOrderSystemInterface
```

Lst. 3.2: Definition einer nicht-funktionalen Eigenschaft in WSML (nach Fensel et al. (2008))

Durch den Algorithmus erfolgt zunächst die Extraktion aller relevanten Informationen aus der Zielbeschreibung (NFP, Ordnungssinn, Instanzdaten, usw.). Danach wird für jeden Dienstkandidat geprüft, ob für die NFP Zusicherungen existieren. Ist dies nicht der Fall, so ist die Bewertung bzw. Gewichtung 0, ansonsten wird sie durch einen Reasoner unter Nutzung der Regeln und Instanzdaten berechnet. Die Gesamtwertung eines Dienstkandidaten ergibt sich aus der Summe aller gewichteten und normalisierten Werte für die geforderten NFP. Die Sortierung aller Kandidaten erfolgt schließlich entsprechend des angegebenen Ordnungssinns.

Die Algorithmen zur Rangfolgebildung sind überwiegend generischer Natur und arbeiten auf der Grundlage semantischer Dienstbeschreibungen. Deshalb sind sie gleichsam gut geeignet für die Anwendung auf Mashup-Komponenten und können in die spätere Konzeption einbezogen werden. Voraussetzung hierfür ist die semantische Annotation bzw. Typisierung von NFP. Relevante Wichtungskriterien in konventionellen, dienstbasierten Systemen umfassen Performanz, Verfügbarkeit, Sicherheit und Skalierbarkeit und klassische QoS-Kriterien, wie Antwortzeiten. Im Hinblick auf Mashup-Anwendungen muss geklärt werden, welche zusätzlichen NFP in die Rangfolgebildung einbezogen werden müssen, um z. B. die Eignung für bestimmte Endgeräte, Interaktionstechniken, Nutzergruppen etc. widerzuspiegeln. Aufgrund der Verortung von SOA im Unternehmensumfeld wird weiterhin von stark formalen Repräsentationen ausgegangen. Die Eignung zur möglichst einfachen Modellierung durch Nicht-Programmierer muss bei der Konzeption in Frage gestellt werden. Schließlich stehen für die dynamische Berechnung von Gewichtungskriterien

in interaktiven Mashups keine Instanzdaten zur Verfügung, die bei der Dienstkomposition häufig den Ausgangspunkt bilden. Das Ranking muss deshalb allein auf Basis der Zielbeschreibung und Daten aus dem Kontextmodell erfolgen, z. B. anhängig von visuellen Anforderungen oder finanziellen Möglichkeiten des Nutzers.

Rekursive Komposition und Ausführung

Rekursive Komposition kommt zum Einsatz, wenn die geforderte Funktionalität unter Berücksichtigung der nicht-funktionalen Anforderungen nicht durch einen, sondern nur durch die Kopplung mehrerer Dienste erreicht werden kann. Wenn diese jeweils nur eine Teilfunktionalität umsetzen, müssen sie möglichst automatisch miteinander verknüpft werden, um die gewünschte Gesamtfunktionalität zu bieten. Einen guten Überblick über existierende Konzepte geben u. a. Dustdar und Schreiner (2005), Rao und Su (2005) und Marconi und Pistore (2009).
Die Ausführung einer Dienstkomposition erfolgt schließlich durch den Aufruf des ersten Dienstes mit den eingehenden Instanzdaten. Der Prozess wird daraufhin vollautomatisch ausgeführt, und das Ergebnis zurückgeliefert. Während i. d. R. ein vollständiger Ablaufplan zugrunde liegt, so kann dieser auch iterativ berechnet werden. Hristoskova et al. (2011) stellen mit ihrem Kompositionframework einen Mechanismus vor, der die Berechnung des Prozesses in Teilschritten bis zum nächsten sicheren Zustand erlaubt. Im Fehlerfall, z. B. bei nicht erreichbaren Diensten, kann der Prozess von diesem Zustand wieder aufgenommen werden und somit einer vollständigen Rekomposition vorbeugen.
Die Übertragbarkeit dieser Konzepte auf interaktive Mashup-Anwendungen ist gering, da sich die rekursive Komposition von Web-Services nur auf die Verknüpfung auf der Datenebene beschränkt. Die Komposition von Benutzerschnittstellen bringt allerdings eine Vielzahl neuer Probleme mit sich, wie die Anordnung der geschachtelten UI-Objekte oder andere visuelle Aspekte, z. B. hinsichtlich einheitlicher Corporate Designs. Zudem ist die vollautomatische Ausführung der Komposition weder möglich noch angestrebt, da die Nutzerinteraktion im Mittelpunkt steht. Auch aus diesem Grund lässt sich kein kausaler Zusammenhang zwischen Ein- und Ausgaben einer Komponente treffen, wie dies bei Web Services der Fall ist.

Mediation

Komposite Anwendungen sind von der Heterogenität ihrer Komponenten und Daten gekennzeichnet. Dieser Heterogenität kann auf Schnittstellenebene durch eine Komponentisierung bzw. ein gemeinsames Komponentenmodell begegnet werden. Auf der Datenebene ist jedoch eine Überführung der verschiedenen Datenschemata nötig. Ähnliche Probleme adressiert der SWS-Nutzungsprozess mit der Mediation.
Trotz semantischer Kompatibilität stellt die syntaktische Heterogenität von Dienstschnittstellen und Daten, d. h. die Nutzung unterschiedlicher struktureller Repräsentationen, das größte Hindernis für die erfolgreiche Ausführung kompositer Dienste dar. Nagarajan et al. (2007) bieten eine umfassende Auflistung der möglichen Probleme hinsichtlich struktureller und semantischer Diskrepanzen verwendeter Daten. Sie unterscheiden zwischen (1) Inkompatibilitäten der Domäne auf Attributebene, wie Namens-, Skalen- und Repräsentationskonflikte bei Datentypen, (2) Inkompatibilitäten von Entitätsdefinitionen, wie Namens- und Isomorphismusprobleme, und (3) Inkompatibilitäten bei der Abstraktion, wie Generalisierungs- und Aggregationsprobleme. All diese Probleme werden im Kontext von SWS durch die

Nutzung von Ontologien adressiert. Dabei spielen u. a. die (semi-) automatische Identifizierung von Ähnlichkeiten und die Abbildung von Ontologien ineinander (*Ontology Alignment* und *Ontology Relation*) eine tragende Rolle.
Für die Mediation wurden eine Reihe von Vorschlägen unterbreitet, z. B. von Fensel et al. (2008) und Nagarajan et al. (2007). Diese sehen die direkte Umwandlung der Daten in das Zielformat vor, was jedoch aufwändige Abbildungsvorschriften mit sich bringt. Deshalb werden u. a. von Corcho et al. (2007) und Szomszor et al. (2006) mehrstufige Transformationen empfohlen, die hinsichtlich Wartung und Skalierung Vorteile bieten, und bei denen generische Reasoning-Mechanismen zum Einsatz kommen. Bei diesen Ansätzen wird das Nachrichtenformat zunächst in eine kanonische Repräsentation bzw. Zwischensyntax (*Lifting*) überführt, woran sich eine optionale Mediation auf semantischer Ebene anschließt, bevor die Transformation in die Zielsyntax (*Lowering*) erfolgt. Diese Zwischensyntax wird für SWS durch Ontologien, z. B. in Form von Resource Description Framework (RDF), beschrieben. Zur Verdeutlichung wird im Folgenden kurz auf die Funktionsweise von SAWSDL eingegangen. Von zentraler Bedeutung sind in der Spezifikation die Attribute *lifting/loweringSchemaMapping*, die Transformationsskripte beliebiger Sprache referenzieren können, um XML-Instanzdokumente zur Laufzeit in Instanzen des annotierten semantischen Konzepts abzubilden, und umgekehrt. Diese Skripte müssen zur Entwicklungszeit auf Schemaebene definiert werden, wobei Sprachen wie eXtensible Stylesheet Language Transformation (XSLT) und SPARQL Query Language for RDF (SPARQL) zum Einsatz kommen. Zur Laufzeit werden sie schließlich auf Instanzebene angewendet. Abbildung 3.7 illustriert das Konzept für den Fall einer gemeinsam genutzten Ontologie. Kommen für Quell- und Zieldaten unterschiedliche semantische Modelle zum Einsatz, müssen zusätzliche Transformationen definiert werden, was jedoch über den Fokus dieser Arbeit hinausgeht.

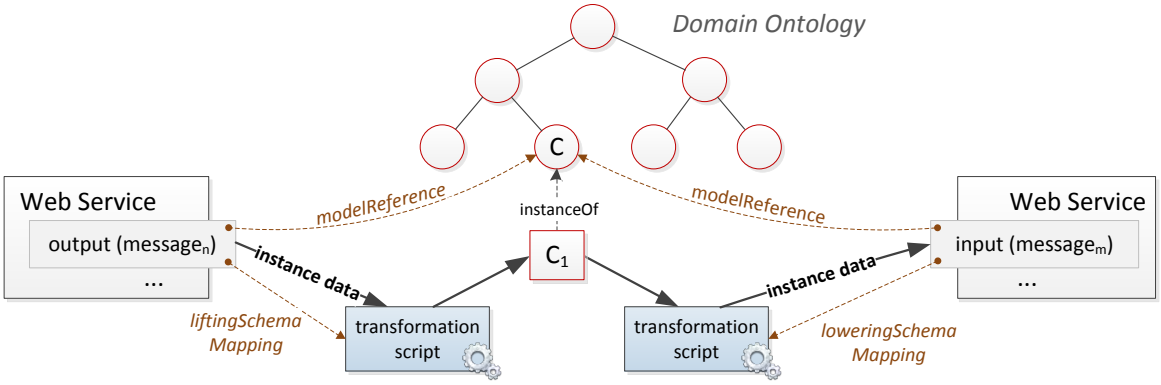

Abb. 3.7: Semantische Mediation von Diensten mit SAWSDL

Weitergehende Ansätze verfolgen das Ziel, die Transformationsvorschriften möglichst vollautomatisch abzuleiten [Bowers und Lüdäscher, 2004; Bleul et al., 2008], um den Aufwand für die manuelle Erstellung zur Entwicklungszeit zu minimieren. Eine generische, automatische Generierung ist jedoch als schwierig bis unmöglich einzuschätzen [Nagarajan et al., 2007], weshalb derartige Ansätze mit einer Reihe von Einschränkungen und Vereinfachungen verbunden sind. Beispielsweise beschränken sich [Bleul et al., 2008] auf die Unterstützung einfacher XML-Schema Datentypen, was für komplexe Web Services und Mashup-Komponenten nicht ausreichend ist.

Fazit

Das Ziel von SWS ist es, den Lebenszyklus von Diensten von der Suche, Sortierung, Komposition bis hin Ausführung möglichst automatisiert durch geeignete Ausführungsumgebungen zu unterstützen. Grundlage hierfür bieten formale Repräsentationen von deren funktionaler und nicht-funktionaler Semantik. Die Komposition in präsentationsorientierten Mashups ähnelt in starker Weise diesem *SWS usage process*. Allerdings ist im Fall von SWS die vollautomatische Erstellung eines Kompositionsplans zur Erfüllung einer spezifischen Aufgabe das Ziel [Arroyo et al., 2004], welche i. d. R. die Verarbeitung bestimmter Eingangsdaten umfasst. Die in dieser Arbeit angestrebte Komposition interaktiver Webanwendungen ist kein solches zustandsbasiertes Planungsproblem (vgl. [Klusch, 2008]), sondern umfasst die kontextsensitive Bindung und Integration von Komponenten, einschließlich der Präsentationsebene, ausgehend von einem semantischen Anwendungsmodell.
Die SWS-Konzepte zur dynamischen, kontextadaptiven Auswahl von Diensten bieten dafür einige Ansatzpunkte. Durch die Abstraktion von konkreten Endpunkten bzw. Komponentenimplementierungen kann deren Auswahl zur Laufzeit erfolgen. Gerade für UI-Komponenten ist dies vorteilhaft, da sie nicht verteilt, sondern auf dem Client-Gerät ausgeführt werden und die entsprechenden Rahmenbedingungen dafür zur Laufzeit überprüft werden können. Die Wahl konkreter Endpunkte bzw. Instanzen erfolgt i. d. R. auf Basis semantisch annotierter Schnittstellen und QoS-Anforderungen bzw. Constraints. Auch die dazu verwendeten Annotationskonzepte können für die Auswahl und Integration interaktiver Komponenten wiederverwendet werden. Allerdings bedarf es dafür Erweiterungen, da letztere nicht entfernt ausgeführt sondern integriert werden, und dafür zusätzliche Aspekte wie Interaktions- und Laufzeitkonzepte beachtet werden müssen. Zudem müssen die Orchestrierungsmodelle auch zeitliche und örtliche Relationen im Sinne von Layouts und Sichten auf die Anwendungen spezifizieren können, was mit den existierenden Ansätzen nicht möglich ist.
Die Methoden zur rekursiven, dynamischen Dienstkomposition lassen sich nur sehr bedingt auf Mashup-Kompositionen anwenden, da auf der Präsentationsebene eine Reihe neuer Probleme betrachtet werden müssen, die sich nicht vollautomatisch berechnen lassen, z. B. die Interaktion mit dem Nutzer, Interaktionsabhängigkeiten und -modalitäten, Look-and-Feel, Layouts, Sichten, etc. Weiterhin kann bei interaktiven Komponenten keine direkte Ein-Ausgabe-Beziehung wie bei konventionellen Diensten unterstellt werden – Konzepte mit Vor- und Nachbedingungen lassen sich folglich nur bedingt anwenden. Gleichermaßen kann nicht von einem vordefinierten Zielzustand ausgegangen werden, der vollautomatisch erreicht werden muss, da die Nutzerinteraktion nicht vorhersehbar ist.
Durch die semantische Beschreibung aller Bestandteile einer Komposition kann die Interoperabilität gesteigert werden, da syntaktische Inkompatibilitäten – wie im letzten Abschnitt beschrieben – dynamisch aufgelöst werden können. Die dazu genutzten Konzepte lassen sich ohne weiteres für interaktive Mashup-Anwendungen nutzen, sofern es sich um Datenmediation handelt. Die zwischen Komponenten ausgetauschten Daten müssen dazu semantisch annotiert werden. Der Ansatz von SAWSDL bietet dahingehend eine hohe Flexibilität, da er mit Anpassungen auch zur Annotation von Mashup-Beschreibungen genutzt werden kann. Zudem sind Transformationssprache und Zielsyntax nicht beschränkt.

3.1.3 Adaptionskonzepte für Dienstkompositionen

Neben der initialen Dienstauswahl und -komposition widmen sich verschiedene Forschungsarbeiten der Frage, wie Kompositionen zur Laufzeit an veränderliche Kontextbedingungen angepasst werden können. Konzeptionell folgen diese Lösungen dem MAPE-Referenzmodell [Miller, 2005], welches eine Kontrollschleife zur Selbstadaption autonomer Systeme beschreibt. Wie Abbildung 3.8 verdeutlicht, durchläuft der Adaptionsprozess von der Erfassung durch Sensordaten bis zur Ausführung konkreter Aktionen durch Effektoren vier Phasen:

Monitoring: Zunächst zeichnen Software- oder Hardware-Sensoren Ereignisse auf, die auf eine Veränderung des Systemzustands, der Plattform oder der Anwendung selbst schließen lassen.

Analysis/Decision: In Folge der Kontextänderung wird auf Basis vordefinierter Funktionen aus der Wissensbasis entschieden, ob eine Reaktion nötig ist.

Planning: Ist eine Änderung nötig, so wird der *Planner* über den gewünschten Zielzustand informiert. Er bestimmt daraufhin alle nötigen Aktionen, um vom aktuellen zum Zielzustand zu gelangen.

Execute: Die abstrakten Aktionen aus der Planungsphasen werden dann in konkrete Aktionen des Systems umgesetzt.

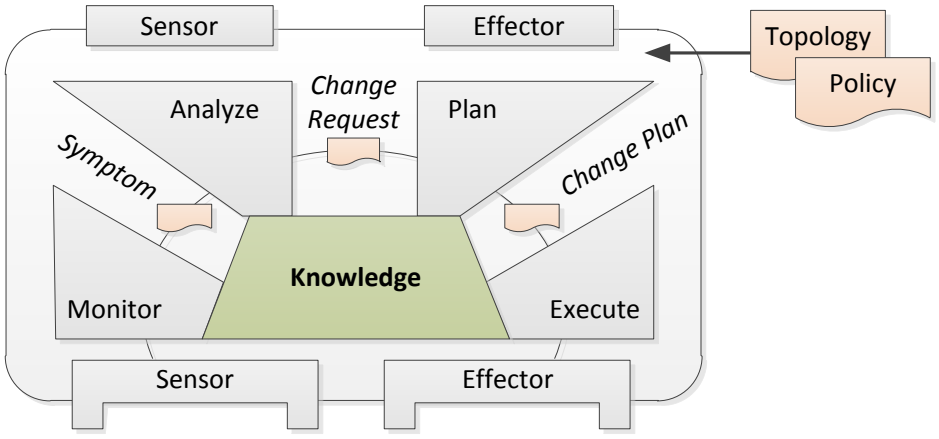

Abb. 3.8: MAPE Referenzmodell für autonome Systeme [Miller, 2005]

Alle Entscheidungen – sowohl für die adaptive Dienstauswahl als auch für die Anpassung zur Laufzeit – werden auf der Grundlagen von Wissen (*Knowledge*) getroffen, welches beispielsweise durch Lösungstopologien (z. B. verfügbare Bestandteile) oder Policies in die Systeme eingebracht wird. Es wird unterschiedlich repräsentiert, z. B. in Form von *utility* und *predictor functions* [Garlan et al., 2009], in Form von Regeln wie SWRL [Hristoskova et al., 2011] oder in der Mehrzahl der Ansätze durch WS-Policy [@Vedamuthu et al., 2007] und Erweiterungen davon [Erradi et al., 2006; Gjørven et al., 2008b].
Für die Adaption selbst existieren verschiedene Herangehensweisen. Die dynamische, aspektorientierte Anpassung, d. h. das Einweben von Funktionalität in Komponenten zur Laufzeit, ist für die Anwendung auf Dienst- und Mashup-Kompositionen wenig geeignet, da sie eine gemeinsame technologische Basis sowie *hooks* innerhalb

der Komponenten voraussetzt. Die Adaption erfolgt deshalb in serviceorientierten Systemen naturgemäß auf der Schnittstellenebene in folgender Art und Weise [André et al., 2010]: Bei der *Parameteradaption* werden Instanzdaten für den Dienstaufruf kontextabhängig verändert; die *funktionale Adaption* erfolgt durch Austausch von Dienstimplementierungen unter Beibehaltung der Schnittstelle; die *Verhaltensadaption* tauscht ebenfalls Dienstimplementierungen aus, was jedoch zur Änderung der Schnittstelle führen kann; die *strukturelle Adaption* ergibt sich durch das Hinzufügen oder Entfernen von Diensten aus der Komposition; die *Umgebungsadaption* betrifft letztlich die Ausführungsumgebung der Anwendung oder der Dienste selbst, z. B. im Fall der Migration.

Die Defizite existierender Konzepte sollen im Folgenden kurz anhand der wsBus-Middleware [Erradi et al., 2006] veranschaulicht werden. Diese setzt die Anpassung von Dienstkompositionen um, wobei auf *Policies* in Form von Event Condition Action (ECA)-Regeln zurückgegriffen wird. Letztere stellen eine Erweiterung von WS-Policy dar und werden auf „virtuelle Endpunkte“ angewendet, die eine Abstraktion für mehrere äquivalente Dienste darstellen. Falls eine Policy verletzt wird oder ein Fehler auftritt, erfolgt die Adaption in Form des erneuten Dienstaufrufs, durch Austausch oder Überspringen des Dienstes.

Abbildung 3.9 veranschaulicht das Gesamtsystem von wsBus in vereinfachter Form mit den wichtigsten Komponenten. Über den *QoS Measurement Service* werden die gängigen Qualitätskriterien von Diensten, wie Zuverlässigkeit, Verfügbarkeit und Antwortzeit, ermittelt. Der *Monitoring Service* ist für die Überprüfung der vordefinierten Policen verantwortlich, und der *Adaptation Manager* entscheidet bei Verletzungen oder Fehlern schließlich, in welcher Form Anpassungen an der Komposition durchzuführen sind. Die Adaptionsmöglichkeiten sind durch die entsprechenden Module erkenntlich: Neben der Auswahl alternativer Dienste, ausgehend von den abstrakten virtuellen Endpunkten, können auch Nachrichten selbst angepasst, d. h. vor- und nachverarbeitet werden.

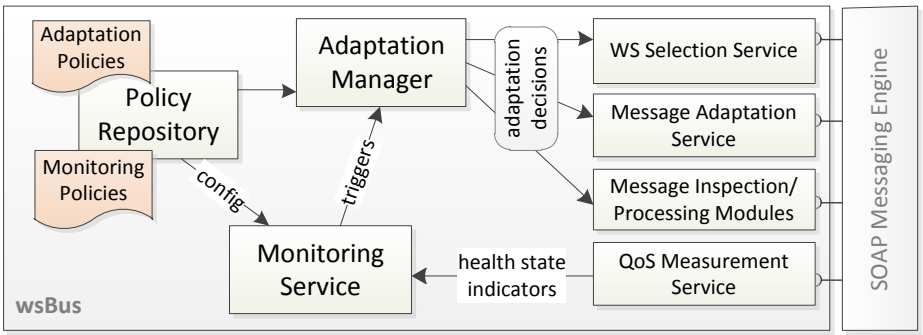

Abb. 3.9: Middleware zur adaptiven Dienstkomposition (nach [Erradi et al., 2006])

Sowohl hinsichtlich der regelbasierten Beschreibung der Adaption als auch architektonisch bietet der Ansatz Konzepte, die in Teilen auf universelle Kompositionen übertragbar sind. Die überwachten QoS-Kriterien und die verwendeten Adaptionstechniken sind für interaktive Mashup-Anwendungen jedoch unzureichend. Da von zustandslosen, verteilt ausgeführten Diensten ausgegangen wird, kommen nur konventionelle Kenngrößen (Verfügbarkeit, Antwortzeit) als Auslöser von Adaption

zum Einsatz, während für die UI-Integration relevante Kontextinformationen, wie Nutzer- und Geräteeigenschaften, freilich keine Rolle spielen.

In Systemen, die umfassenderes Kontextwissen in die Adaption einbeziehen, werden i. d. R. ontologiebasierte Modelle eingesetzt. Neben der starken Formalisierung bieten semantische Technologien Vorteile bei der Konsistenzsicherung und Ableitung neuen Wissens. Eine ausführliche Darstellung der Funktionsweise eines typischen Vertreters bieten Y. Li et al. (2008), welche einen Ansatz zur kontextsensitiven Dienstkomposition unter Nutzung der Top-Level-Ontologie *PLATE* vorstellen.

Im Hinblick auf das zu entwickelnde Konzept stellt der Zustand von Komponenten eine besondere Herausforderung bei der Adaption dar – insbesondere bei deren Austausch. Im Gegensatz zur Mehrzahl existierender Systeme widmen sich Irmert et al. (2008) mit dem *Component Based Runtime Adaptable* (CoBRA) Framework genau diesem Problem. Dienste werden darin durch Komponenten erbracht, die sowohl zustandslos als auch zustandsbehaftet sein können. Der Zustand drückt sich durch eine Reihe von Variablen aus, die gemäß dem *Memento*-Muster [Gamma et al., 1995] beim Tausch von der ursprünglichen auf die ersetzende Komponente übertragen werden. Dazu müssen die Komponenten explizite Methoden umsetzen, um ihre zustandsgebenden Variablen in einen separaten Container zu speichern und von dort zu laden. Um in der zwischenzeitlichen Phase des Austauschs (*switch*) Inkonsistenzen und Nachrichtenverluste zu vermeiden, wird das Prinzip der *Protection Proxies* [Gamma et al., 1995] angewendet. Dabei wird jeder Dienst der Komposition – für die Dienstnutzer transparent – durch jeweils eine lokale Proxy-Instanz abgeschirmt, die jegliche Aufrufe bis zur Beendigung des Austauschs sichert, und danach an die neue Komponente delegiert. Die verwendeten Muster sind generisch und können gleichermaßen bei der Adaption universeller Kompositionen zum Einsatz kommen.

Fazit

Alle eingangs genannten Adaptionstechniken sind ebenso für universelle Kompositionen nötig und möglich, wobei in dieser Arbeit insbesondere die funktionale und Parameteradaption in Vordergrund stehen. Zur Adaption wird auf die Umsetzung des *MAPE*-Modells bzw. einer externen Kontrollschleife zur Einhaltung vordefinierter Kriterien gesetzt, wie in FAMOUS [Nilsson et al., 2006] und ACCADA [Gui et al., 2009]. Die gemeinsame Anstrengung richtet sich bei diesen Ansätzen auf die kontextadaptive Erstellung von Kompositionsplänen und Nutzenfunktionen (*utility functions*). Die entsprechenden Middlewares unterstützen nur wenige vordefinierte Adaptionsmechanismen, die zudem plattformspezifisch realisiert sind [Gjørven et al., 2008a]. Kein System unterstützt beispielsweise gleichzeitig die Parameteradaption, die funktionale Adaption und die adaptive (Re-)Komposition [Treiber et al., 2010].

Neben der Schnittstelle zwischen Plattform und Komponenten bzw. Diensten muss jedoch auch die Schnittstelle zum Nutzer beachtet werden. In interaktiven Anwendungen bedarf gerade diese der Anpassung, was eine Vielzahl neuer Herausforderungen (adaptives Layout, adaptive Interaktionstechniken, adaptive Aufteilung auf *Screens* bzw. Sichten, usw.) mit sich bringt, auf die bisherige Konzepte aus dem SOA-Umfeld Antworten schuldig bleiben. Dies gilt stellvertretend für die breite Masse an Literatur im Bereich der dynamischen Service-Komposition und -Adaption. Anpassungen auf der Präsentationsebene, wie sie klassische Modelle aus dem Bereich des Adaptive Hypermedia betrachten (vgl. Abschnitt 3.2) sind mit ihren Mitteln nicht abbildbar.

3.1.4 Interaktions- und UI-Konzepte für Dienstkompositionen

Über die Konzepte von B4P und WS-HT hinaus (Abschnitt 3.1.1) ist die Anzahl der Forschungsansätze, die sich mit der Interaktion mit Diensten und der Bereitstellung zugehöriger Benutzerschnittstellen beschäftigt, im Bereich SOA sehr gering.

Symptomatisch ist die Arbeit von Treiber et al. (2010), die die Kombination klassischer *Software-* und neuartiger *Human-Provided Services* (SPS, HPS) betonen. Durch ein spezifisches Kontextmodell und ein vordefiniertes QoS-Modell kann das Prozessmodell beeinflusst werden: Kontextadaptivität bezieht sich in dem Fall auf die Auswahl von Aktivitäten und dafür vordefinierte Dienste. Die Abbildung menschlicher Interaktionen in Form von HPS ist jedoch weitestgehend mit dem bereits vorgestellten Konzept WS-HumanTask vergleichbar, d. h. Gestaltung, Layout, Komponentisierung und Komposition auf der Präsentationsebene werden nicht modelliert. Konzepte zu deren Anpassung finden dementsprechend auch keine Anwendung.

Die Nutzung von **Dienstannotationen** zur Generierung kontextspezifischer Oberflächen bietet dahingehend einen größeren Spielraum. Sie kann einerseits der impliziten Verknüpfung der darunter liegenden Dienste dienen (*Komposition durch UI*), und andererseits der Erstellung integrierter Oberflächen für existierende funktionale Dienstkompositionen (*UI für Komposition*).

Im Projekt ServFace [Feldmann et al., 2009] [Dannecker et al., 2010] wird das erste Ziel verfolgt. Dazu wird die WSDL-Beschreibung jedes Dienstes um ein Annotationsdokument erweitert, welches u. a. Informationen zur visuellen Erscheinung und zum Verhalten der einzelnen Dienst-Operationen besitzt. Wie Abbildung 3.10 zeigt, erlauben derartig publizierte Dienstbeschreibungen die Kopplung von Diensten auf der Präsentationsebene mittels geeigneter Entwicklungswerkzeuge, wie dem ServFace Builder [Nestler et al., 2009]. Der Entwickler verknüpft dazu *service frontends*, die Operationen von SOAP-Diensten repräsentieren und für die Ein- und Ausgabe der benötigten Parameter UI-Elemente bieten, welche aus den Annotationen generiert werden. Eine abgeschlossene Komposition wird durch das plattformunabhängige *Composite Application Model* (CAM) beschrieben, das durch Model-to-Code-Transformationen in ausführbare Anwendungen für potentiell verschiedene Plattformen transformiert werden kann. Alternativ können zur Repräsentation der Anwendung auch Task-Modelle auf Basis der ConcurTaskTree (CTT)-Notation genutzt werden, die um ein UI-Modell in *MARIA* [Paternò et al., 2009] angereichert sind.

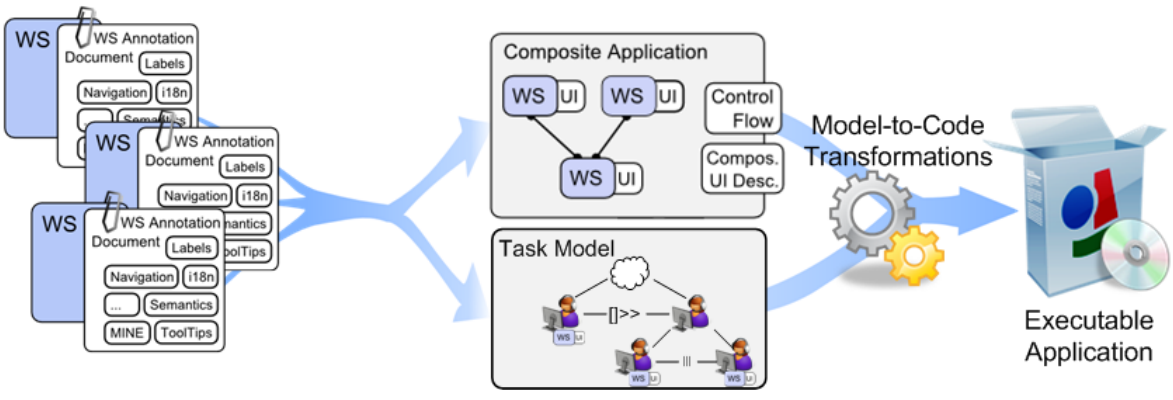

Abb. 3.10: Visuelle Web-Service-Komposition in ServFace [@ServFace, 2011]

Im Vordergrund steht bei ServFace die Komposition von SOAP-Diensten durch Endnutzer, weniger die Entwicklung einer RIA-Oberfläche. Die generierten Nutzerschnittstellen sind entsprechend einfach gehalten und eher Mittel zum Zweck als Teil der Komposition selbst. Das Komponentenmodell folgt den Charakteristika der SOAP-Dienste und geht von zustandslosen Bestandteilen aus, deren Kommunikation sich auf den unidirektionalen Datenfluss zwischen Operationen beschränkt. Konzepte für die dynamische Bindung oder Adaption der Komposition und insbesondere der Benutzerschnittstelle sind nicht vorgesehen.

Kateros et al. (2008) stellen einen weiteren annotationsbasierten Ansatz vor, allerdings bezieht sich die Annotation hier auf das Kompositionsmodell. Dienste werden dabei mittels Unified Modeling Language (UML) auf Basis eines eigenen Modells repräsentiert, welches problemlos aus bestehenden WSDL-Beschreibungen generiert werden kann. Durch die Modellierung von *FrontEndViews* (Zustände) und *FrontEndActions* (Auslöser für Zustandsübergänge) durch den Entwickler ist es möglich, die Benutzerschnittstelle zu beschreiben. Dies geschieht mit Hilfe von Tags, allerdings auf sehr niedriger Abstraktionsebene, z. B. in Form von *label*-, *button*- und *hyperlink*-Annotationen. Aus dem fertigen Modell wird schließlich plattformspezifischer Code generiert. Aufgrund der beschränkten Annotationsmöglichkeiten ist die generierte Oberfläche sehr rudimentär und stellt nach Aussage der Autoren nur den Ausgangspunkt für UI-Designer dar. Dem Problem der aufwändigen UI-Entwicklung wird mit dem Konzept also nicht begegnet.

Auch [Achilleos et al., 2011] widmen sich der Interaktion mit serviceorientierten Anwendungen und stellen für diese einen modellgetriebenen Entwicklungsprozess vor, der die Trennung von Präsentations- und Kompositionsschicht vorsieht. Für alle Dienste werden automatisch plattformspezifische Zugriffsklassen (*Service Clients*) generiert, und der Entwickler definiert parallel ein abstraktes Präsentationsmodell auf Basis der Presentation Modeling Language (PML). Dies geschieht, wie beim vorherigen Ansatz, auf einer sehr niedrigen, plattformspezifischen Abstraktionsebene, indem konkrete Elemente, Modalitäten usw. angegeben werden. Details zur Kopplung von PML- und Service-Modell bleiben die Autoren schuldig. Letztlich handelt es sich um ein aufgesetztes UI-Modell, welches kein Teil der eigentlichen Komposition ist und somit den identifizierten Anforderungen, z. B. hinsichtlich später Bindung oder Adaptivität, nicht gerecht wird.

Visionär kann mit Blick auf die Ziele dieser Arbeit das SOAUI-Konzept [Tsai et al., 2008] bezeichnet werden, welches die konsequente Anwendung der Dienstprinzipien auf Benutzerschnittstellen propagiert und durch Abbildung 3.11 veranschaulicht wird. Anwendungen werden durch *Application Templates* beschrieben, die sog. *UI composition points* beinhalten können. Letztere entsprechen Templates im Sinne der semantischen Dienstsuche und dienen zur Laufzeit als Ausgangspunkt bei der Suche nach User Interface Services (UISs) in der *Service Registry*. UIS stellen Benutzerschnittstellen bzw. Teile davon bereit, werden allerdings vollständig deklarativ, z. B. in eXtensible Application Markup Language (XAML) oder User Interface Modeling Language (UIML), spezifiziert. Sind alle Composition Points durch passende Dienste gebunden, ist die Anwendungsspezifikation komplettiert. Das entstandene Anwendungsmodell – beispielsweise in XAML kann nun gerendert bzw. in ausführbaren Code umgewandelt werden.

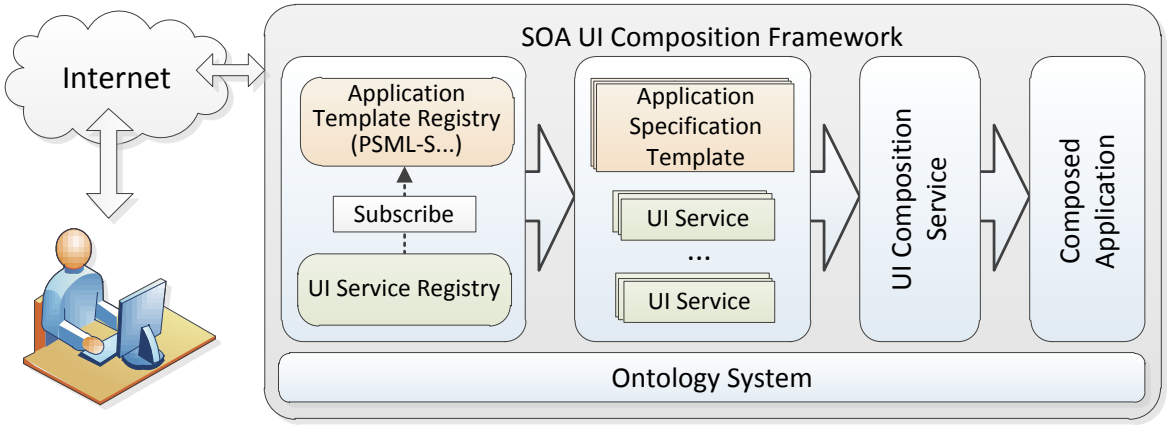

Abb. 3.11: SOAUI-Konzept zur dienstbasierten UI-Komposition [Tsai et al., 2008]

Der Ansatz zeigt, wie sich das Dienstprinzip samt später Bindung auf der Präsentationsebene umsetzen lässt. Allerdings unterliegt er einer Vielzahl von Einschränkungen. Zum einen wird von einer offenen, deklarativen Spezifikation aller Komponenten in einer gemeinsamen Technologie ausgegangen. Zum anderen ist die Komposition allein auf die Präsentationsebene beschränkt – die gegenseitige Integration mit Web Services ist nicht vorgesehen. Die Nutzung von Kontextinformationen im Kompositionsprozess oder zur Laufzeit ist ebenfalls nicht Teil des Konzeptes.

Fazit

Es ist ersichtlich, dass die vorgestellten Konzepte zur Interaktion mit Dienstkompositionen Beschränkungen unterliegen, die ihrer Nutzung zur Entwicklung interaktiver Mashup-Anwendungen entgegenstehen. Grundsätzlich ist festzustellen, dass die Benutzerschnittstelle in keinem der Ansätze ein Teil der Komposition selbst ist, sondern entweder auf Basis von Annotationen generiert, oder separat modelliert wird. Es fehlt somit an einem ganzheitlichen Konzept zur universellen Komposition, wie es in dieser Arbeit angestrebt wird. Dies hat zwei mögliche Auswirkungen: Kommen relativ einfache UI-Modelle oder -Annotationen zum Einsatz, so äußert sich dies i. d. R. in beschränkten, generierten Oberflächen. Mit steigender Komplexität und sinkender Granularität können reichhaltigere Benutzerschnittstellen erstellt werden – gleichzeitig steigen Entwicklungsaufwand und Plattformabhängigkeit der Beschreibung. Ferner mangelt es den Lösungen an Konzepten zur dynamischen Bindung und Kontextadaptivität.

3.2 Web Engineering - Entwicklung interaktiver adaptiver Webanwendungen

Bereits zu Beginn der 1990er Jahre war klar, dass die Entwicklung von Webanwendungen kein singuläres Ereignis, sondern ein fortwährender Software-Entwicklungsprozess ist [Garzotto et al., 1993]. Dementsprechend wurden diverse Modelle vorgestellt, die die strukturierte Erstellung komplexer Webanwendungen erlauben. Durch die in Kapitel 1 beschriebene Wandlung des Internets zu einer Anwendungsplattform haben sich jedoch die damit verbundenen Charakteristika

und Technologien nachhaltig geändert. Die Veränderung von klassischen Web-Engineering-Ansätzen zu Mashups ist dabei vergleichbar mit der Evolution von der objektorientierten zur komponentenorientierten Softwareentwicklung. Die Mashup-Entwicklung geht i. d. R. nicht von einem mentalen (und formalisierten) Modell der (Daten-)Objekte aus – vielmehr steht die Kombination von *Funktionalitäten* in Form von vorgefertigten Bausteinen im Vordergrund.

Die folgenden Abschnitte beleuchten existierende Entwicklungsansätze und Architekturen aus beiden Bereichen und diskutieren ihre Eignung zur Modellierung adaptiver interaktiver Webanwendungen im Sinne dieser Arbeit.

3.2.1 Entwicklung von Hypertext- und Hypermedia-Anwendungen

Seit mehr als einem Jahrzehnt widmet sich die Web-Engineering-Community Entwicklungsprozessen und -modellen für Hypermedia-Anwendungen. Wright und Dietrich (2008) evaluierten die verbreitetsten Ansätze hinsichtlich ihrer Eignung zur Modellierung von RIAs. Dazu wurden Kriterien, wie die Plattformunabhängigkeit bei der Entwicklung, die Unterstützung von Standards, die Nutzung von Metamodellen, aber auch der Reichhaltigkeit der angebotenen Oberflächen analysiert, die auch im Rahmen dieser Arbeit einen hohen Stellenwert besitzen.

Konventionelle Ansätze strukturieren Anwendungen demnach durch Modelle, die von festen Datenschemata und -konzepten ausgehen. Präsentationsmodelle beschreiben die Oberflächen zur Darstellung der Inhalte, wobei fast ausschließlich deklarative Beschreibungssprachen zum Einsatz kommen, die Ausgangspunkt für die Generierung von HyperText Markup Language (HTML)-Markup sind. Die Unterstützung von Interaktivität beschränkt sich i. d. R. auf die Definition eines Navigationsmodells, welches Konzepte auf verschiedenen Seiten anordnet und zwischen ihnen *Links* aufbaut. Verteilte Datenmodelle, vorgefertigte UI-Komponenten, reichhaltige Interaktionskonzepte wie Drag-and-Drop, ereignisorientierte Kommunikation – all jene Aspekte werden durch bisherige Lösungen im Hypermedia-Bereich nur unzureichend adressiert [Wright und Dietrich, 2008].

Als Vertreter mit der größten Verbreitung werden im Folgenden WebML [Brambilla et al., 2008] und UWE [Knapp et al., 2007] stellvertretend etwas näher vorgestellt.

WebML [Brambilla et al., 2008] beschreibt sowohl eine visuelle Modellierungssprache als auch einen Entwicklungsprozess. Letzterer beinhaltet verschiedene Modelle, die die Struktur, Komposition, Navigation und Präsentation von Webanwendungen repräsentieren. Die Entwicklung erfolgt datenzentriert auf Basis der strukturellen Modelle, die z. B. in Form von Entity-Relationship- oder UML-Diagrammen vorliegen können und aus denen das grundlegende Datenmodell generiert wird. Die weitere Modellierung fasst die Datenkonzepte zu Daten- bzw. *Content Units*, *Areas* und *Pages* zusammen. Im *Hypertext-Modell* wird schließlich die Navigation definiert, indem Konzepte durch *Links* miteinander verbunden werden. Diese können entweder kontextbehaftet sein, d. h. sie transportieren Daten, oder sie sind kontextfrei und werden auf Hyperlinks abgebildet. Durch die Erweiterungen von Manolescu et al. (2005) ist die Einbindung von Web Services möglich. *Service Units* können dann analog zu bestehenden *Content Units* genutzt und in die Komposition eingebunden werden. Die Funktionalität beschränkt sich freilich auf klassische SOAP-Dienste.

Gängige Metamodell-Standards und Werkzeuge, z. B. zur formalen Modellverifikation, werden nicht unterstützt.
WebML besitzt kein Komponentenmodell im Sinne von CBSE. Die Modellierung erfolgt allein ausgehend von Datenschemata, während die UI mittels XSLT-Stylesheets generiert wird. Die Ausdrucksfähigkeit auf der Präsentationsebene ist folglich stark eingeschränkt und auf die Abbildung in HTML ausgerichtet. In [Preciado et al., 2007] wird deshalb die Integration des durch Linaje et al. (2007) ausführlich vorgestellten RUX-Prozesses in WebML präsentiert. Hinter RUX verbirgt sich die schrittweise Transformation einer abstrakten Nutzerschnittstelle auf eine konkrete, plattformspezifische Repräsentation. In Kombination mit WebML ersetzt RUX dessen Präsentationsmodell, d. h. die Hypertextmodelle von WebML werden direkt mit dem RUX Interface-Modell verknüpft, wie in Abbildung 3.12 angedeutet.

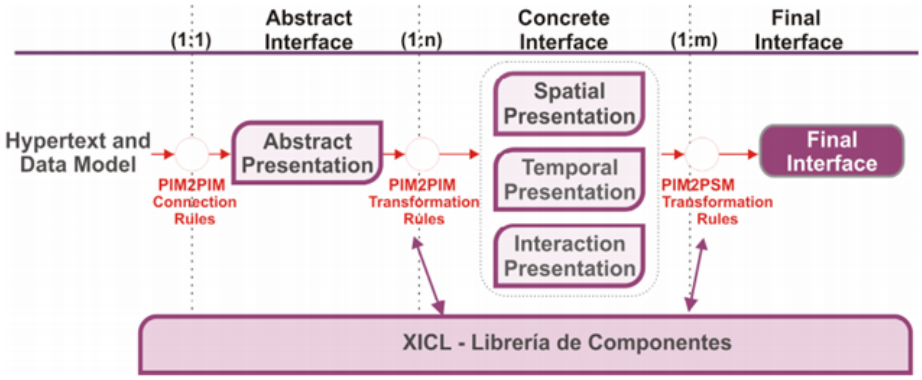

Abb. 3.12: UI-Generierung mit RUX [Preciado et al., 2007]

Dieses Vorgehen erlaubt die Nutzung von Komponenten auf der Präsentationsebene. Diese sind im Fall von RUX allerdings nicht abgeschlossen (*black box*) oder lose gekoppelt, sondern auf sehr niedriger Abstraktionsebene deklariert. Von einer universellen Komposition kann nicht gesprochen werden. Zudem findet der Abbildungsprozess auf plattformspezifische UIs zur Entwicklungszeit statt, d. h. der Entwickler muss um Eigenheiten und Anforderungen der Zielplattform wissen und wählt Komponenten selbst aus. Ein *Late Binding* im Sinne der dynamischen Auswahl passender Komponenten zur Laufzeit ist nicht vorgesehen.
Weiterhin mangelt es WebML zunächst an Konzepten zur Adaption bzw. Kontextsensitivität. Deshalb stellen Ceri et al. (2007) eine Lösung vor, welche die Erweiterung des Datenmodells um ein Nutzer-, ein Personalisierungs- und ein Kontextschema vorsieht. *Pages* können als kontextsensitiv deklariert werden, was zu ihrer Anpassung durch sog. *context clouds* vor dem Aufruf führt. Die Adaptionsmittel beschränken sich folglich auf die Oberfläche und adressiere nicht die Gesamtkomposition (i. S. v. Datenfluss, Sichten, Komponententausch). Zudem setzt das Konzept voraus, dass Inhalte vorher mit Metadaten annotiert werden, wovon bei der Kombination verteilter Dienste von Dritten, wie in Mashups, nicht ausgegangen werden kann[2].

Der modellgetriebene Entwicklungsprozess von UML-based Web Engineering (UWE) [Baumeister et al., 2005; Knapp et al., 2007] weist große Ähnlichkeit zu WebML auf, nutzt zur Modellierung jedoch vollständig UML und ein eigenes UML-Profil. Auch hier

[2]Dies ist als *Closed-Corpus*-Problem [Brusilovsky, 2001] bekannt.

werden die verschiedenen Belange einer Webanwendung, wie *Requirements*, *Content*, *Navigation*, *Presentation* und *Deployment*, durch eigene Modelle repräsentiert.

Eine besondere Bedeutung kommt bei UWE der Modellierung von Adaptivität zu. Sie erfolgt nach den Prinzipien der Aspektorientierung, die durch Kiczales et al. (1997) ausführlich erörtert werden. Mittels *Pointcuts* werden über Muster Modellbestandteile adressiert, die kontextabhängig durch *Advices* verändert werden sollen. Dies hat den Vorteil, dass die Adaptionslogik vollständig von der Anwendung getrennt bleibt und unabhängig entwickelt, getestet und gewartet werden kann. Es wird zwischen *Modellaspekten* und *Laufzeitaspekten* unterschieden. Erstere werden durch einen Aspektweber bereits zur Entwicklungszeit in die Modelle integriert, können also nicht auf den situativen Kontext reagieren. Die Auswertung von Laufzeitaspekten erfolgt hingegen während der Ausführung der Anwendung und dient z. B. zum Hinzufügen oder Annotieren von Links. Wie und in welcher Art die Auswertung und Adaption ablaufen, wird leider nicht näher erläutert.

Die Modelle von UWE sind sehr komplex, dafür aber plattformunabhängig und standardisiert. Leider geht auch UWE nicht von komponenten- bzw. dienstbasierten Anwendungen aus, sondern setzt insbesondere auf der Präsentationsebene eine Modellierung auf niedrigem Abstraktionsniveau ein, z. B. mit Konzepten wie *Text*, *Image* oder *Button*. Die geschaffenen Oberflächen sind deshalb kaum wiederverwendbar. Auch für UWE wurde eine Erweiterung bzw. Kopplung mit RUX vorgestellt [Preciado et al., 2008], die den gleichen Problemen wie bei WebML unterliegt. Die aspektorientierte Modellierung adaptiven Verhaltens ist für das zu entwickelnde Konzept jedoch vielversprechend, auch wenn die Konzepte auf komposite Webanwendungen übertragen werden müssen.

Einige Ansätze konzentrieren sich insbesondere auf die Modellierung reichhaltiger Oberflächen i. S. v. RIAs. Ein typischer Vertreter ist OOH4RIA [Meliá et al., 2008], der die OOH-Methodologie [Gómez et al., 2001] erweitert. Wie die vorherigen Ansätze ist auch OOH4RIA datenorientiert und wenig für die Komposition verteilter Anwendungsbestandteile mit potentiell eigenen Datenmodellen geeignet. Auf der Präsentationsebene kommen *Widgets* als Modellkonstrukte zum Einsatz, die hierarchisch strukturiert werden können. Diese sind zwar aus Nutzersicht reichhaltig, die Entwicklung erfolgt aber auf sehr niedrigem Abstraktionsniveau auf der Ebene von Buttons und Eingabefeldern. Das Präsentationsmodell wird zudem direkt auf das Google Web Toolkit (GWT) abgebildet, ist hinsichtlich Mächtigkeit und Funktionalität stark daran ausgerichtet und als plattformspezifisch zu bewerten. Die Integration von Web Services oder die späte, kontextadaptive Bindung von Diensten oder UI-Elementen ist nicht vorgesehen. Zur Modellierung adaptiver Oberflächen wird in [Garrigós et al., 2009] eine Erweiterung vorgeschlagen, die in einem zusätzlichen *User Model* Regeln zur Veränderung der Widgets enthält. Das Adaptionskonzept beschränkt sich jedoch auf die Präsentationsebene. Im Wesentlichen werden Oberflächenelemente verändert – es erfolgt jedoch keine grundlegende Rekonfiguration oder Rekomposition der Anwendung, z. B. zur Änderung des Daten- oder Kontrollflusses.

Der Unterstützung von Adaptivität für modell- und komponentenbasierte Oberflächen widmen sich auch wenige eigenständige Konzepte.

Grundlage für den Ansatz von Nilsson et al. (2006) bietet das Komponenten-Framework des FAMOUS-Projekts. Im Anwendungsmodell können Komponenten

abstrakt bzw. als Variationspunkte definiert werden, was die dynamische Bindung verschiedener, allerdings vordefinierter Implementierungen ermöglicht. Für alle Variationen werden bereits vor der Ausführung alternative Instanzhierarchien von der Plattform berechnet. Jede Komponente besitzt spezifische *properties*, die im Zusammenspiel mit globalen *utility functions* die Berechnung der besten Alternative durch die Plattform erlauben. Hinsichtlich der Wahl der Mittel ist die Ähnlichkeit zu adaptiven Workflow-Systemen wie reMash! (Angabe alternativer Dienstimplementierungen, vgl. Abschnitt 3.1.2) und selbst-adaptiven SOAs (Nutzung von *utility functions*, vgl. Abschnitt 3.1.3) zu erkennen. Das Konzept ist jedoch auf die Präsentationsebene und die Auswahl aus vordefinierten Varianten beschränkt, deren Schnittstellen zudem syntaktisch äquivalent sein müssen.
Für die UI-Adaption in RIAs schlagen Schmidt et al. (2009) eine Adaptionsschleife vor, die auf einer formalen semantischen Repräsentation der Anwendung aufbaut. Dazu müssen deren Struktur und Inhalte zunächst semantisch annotiert werden. Mittels Web Usage Mining, z. B. unter Nutzung der *Access Logs*, kann das System Nutzungsmuster ableiten, die der Erstellung deklarativer Adaptionsregeln in SWRL durch Domänenexperten dienen. Diese Regeln steuern die Ausführung der Anwendung zur Laufzeit, sind allerdings auf sehr niedrigem Abstraktionsniveau definiert, da sie auf atomaren Konzepten der RIA-Ontologie, wie *Button* oder *Label* arbeiten. Der Ansatz geht, wie die Mehrzahl der vorgestellten Konzepte aus dem Web-Engineering-Bereich, von einer „closed-corpus"-Annahme und somit vorher bekannten Inhalten aus, und nicht von verteilten, lose gekoppelten Komponenten.

Fazit

Zusammenfassend ist festzustellen, dass die konventionellen Web-Engineering-Ansätze der Modellierung und Aggregation von Information auf Basis vordefinierter Domänenmodelle dienen und für die Modellierung und Integration von verteilten Funktionalitäten und Daten mit ggf. verschiedenen Datenmodellen wenig geeignet sind. Abgesehen von Erweiterungen zur Einbindung von Web Services, finden die Prinzipien komponenten- und dienstbasierter Systeme in den vorgestellten Lösungen keine Anwendung. Insbesondere Benutzeroberflächen werden nur deklarativ und auf niedrigem Abstraktionsniveau spezifiziert. Die kontextabhängige Auswahl und Integration von Anwendungsbestandteilen ist mit keinem der Systeme möglich – lediglich Anpassungen der Hypermedia-Oberflächen zur Laufzeit können durch einige Ansätze modelliert werden. Die Adaptionskonzepte beschränken sich jedoch auf vordefinierte Varianten, unterstellen bestimmte Ausführungsplattformen, feste Datenmodelle und vorannotierte Inhalte.

3.2.2 Entwicklung von Mashup-Anwendungen

Im Gegensatz zu den bislang vorgestellten Konzepten impliziert die Modellierung von Mashups die Komposition verteilter Ressourcen bzw. Dienste. Die Forschung in diesem Bereich widmet sich zwei grundlegenden Herausforderungen: (1) der Überwindung von Kompositionsproblemen, wie fehlender Interoperabilität und dynamischer Konfiguration von Komponenten, sowie (2) der Entwicklung endnutzertauglicher Kompositionsmetaphern und -werkzeuge. Die in dieser Arbeit entwickelten Konzepte können dem ersten Themenkomplex zugeordnet werden.

Entsprechende Forschungsansätze adressieren für gewöhnlich spezifische Aspekte, wie die Datenintegration, Service-Komposition, Interaktion zwischen Mashup-Komponenten oder die Interaktion mit Nutzern [Abiteboul et al., 2008]. Die vorliegende Arbeit stellt einen Querschnitt all dieser Themen dar, da sie sowohl die Modellierung als auch die Integration und Ausführung von Mashup-Anwendungen betrachtet. Der Aspekt der Adaptivität stellt einen zusätzlichen, querschneidenden Belang dar.

Portale, die ältesten Vertreter webbasierter Anwendungsintegration, kommen der Mashup-Idee sehr nahe. Sie dienten zunächst der Aggregation von Funktionalitäten bzw. Anwendungen, die typischerweise keine Koordination oder Transaktion benötigten. Inzwischen ist die Technologie gereift und unterstützt im aktuellen Standard (JSR 286 [@Heppner, 2008]) die Kommunikation zwischen Portlets. Die Integration simuliert jedoch nur eine Serviceorientierung [Beemer und Gregg, 2009] – die Anbindung und Integration von Diensten über Portlets erfolgt rein programmatisch. Eine modellgetriebene, plattformunabhängige Entwicklung von Portalanwendungen ist nicht möglich, da man trotz aller Standardisierungsbemühungen aufgrund proprietärer Erweiterungen an einen Hersteller gebunden ist. Generell ist die Interoperabilität von Portlets und Portalen stark beschränkt [Abiteboul et al., 2008]. Dies gilt auch für die Verknüpfung innerhalb einer Portalanwendung, da sich die Kommunikation auf die Weiterleitung von Events beschränkt und den Anforderungen von RIA-Oberflächen, z. B. hinsichtlich der Synchronisation von Komponenten, kaum gerecht wird. Gegenüber Mashups sind Portale deutlich „schwergewichtiger“, d. h. sie benötigen eine umfangreiche Infrastruktur und implizieren die Entwicklung plattformspezifischer, komplexer Portlets.

Die weitere, fokussierte Betrachtung von Mashup-Ansätzen macht eine genauere Einordnung und Charakterisierung dieser nötig. Aufgrund der Nähe zur Dienstkomposition lassen sich auch Mashup-Anwendungen gemäß der Klassifikation aus Abbildung 3.1 nach Fluegge et al. (2006) unterteilen:

Statische Mashups sehen eine manuelle Erstellung vor. Diese kann zum einen programmatisch durch dedizierte Skriptsprachen oder DSLs erfolgen, wie bei IBM WebSphere sMash [@IBM, 2011c] oder dem Google Mashup Editor [@Google, 2011a]. Zum anderen kommen Spreadsheet-basierte Metaphern (StrikeIron Express, Extensio Excel Extender [@Extensio, 2011]) oder visuelle Werkzeuge zur „Verdrahtung“ von Komponenten (*Wiring*) zum Einsatz, wie Yahoo! Pipes [@Yahoo!, 2011a], JackBe Presto Wires [@JackBe, 2011] oder IBM InfoSphere MashupHub [@IBM, 2011a] als Teil des IBM Mashup Centers.

Semi-dynamische Mashups verfügen über Mechanismen, die dem Entwickler weitere Arbeit bei der Komposition abnehmen. So werden von entsprechenden Plattformen proaktiv neue Komponenten vorgeschlagen, erst zur Laufzeit gebunden oder gänzlich automatisch integriert. Die Grenze zu statischen Mashups ist dabei freilich fließend. Zu den typischen Vertretern zählen Karma [Tuchinda et al., 2008] und der Intel MashMaker [Ennals und Trushkowsky, 2008], die das *Programming by Example* ermöglichen.

Dynamische Mashups erlauben letztlich die automatische Dienstaggregation ohne jegliche Nutzerinteraktion, wie in durch Carlson et al. (2008) oder T. Fischer et al. (2009) beschrieben.

Diese Dreiteilung korrespondiert grob mit der Einordnung nach dem Abstraktionslevel bzw. der anvisierten Zielgruppe [Grammel und Storey, 2010; T. Fischer et al., 2009]. **High-Level** Mashups arbeiten auf hohem Abstraktionsniveau und sind eher für Endanwender geeignet. **Intermediate-Level** Mashups setzen ein gewisses Verständnis der zugrunde liegenden Datenstrukturen und der Datenflusskonzepte voraus und sind auf Domänenexperten ausgerichtet. **Low-Level** Mashups bieten die größtmögliche Flexibilität, sehen aber die Entwicklung durch Programmierer vor.
Im Hinblick auf die Zielstellung dieser Arbeit liegt der Fokus der Betrachtung auf der jeweils mittleren Ausprägung: Es soll ein klassischer Entwicklungsprozess unterstützt werden, der die Trennung von Design- und Laufzeit vorsieht, aber Aspekte der Plattformunabhängigkeit und Kontextualität beinhaltet. Insofern scheiden statische, nicht-adaptive Mashups zunächst aus, da sie keinerlei Adaptionskonzepte vorsehen und sich an Programmierer richten. Dynamische, vollautomatische Mashups für Endnutzer entstehen hingegen erst zur Laufzeit und sind eher für das EUD geeignet, als für die strukturierte Anwendungsentwicklung.

Die folgenden Abschnitte widmen sich der Vorstellung und Bewertung existierender Mashup-Ansätze in drei Schritten. Die Betrachtung erfolgt ausgehend von umfassenden Architektur- und Referenzmodellen, legt den Schwerpunkt danach auf grundlegende Konzepte der Modellierung am Beispiel statischer Mashup-Anwendungen, und geht schließlich auf Möglichkeiten der (semi-)dynamischen Komposition, Integration und Adaption ein.

3.2.2.1 Referenzmodelle

Mit Blick auf mögliche Architektur- und Kompositionskonzepte lohnt die Betrachtung existierender Referenzmodelle für Mashups.
Bereits 2006 charakterisierten Jhingran (2006) die *Mashup Fabric* als mehrstufiges Modell, welches vorrangig die Datenaggregation und Integration unter einer eigenen Präsentationsschicht beschreibt. Im Mittelpunkt des Konzeptes steht die leichtgewichtige Informationsintegration durch die Verknüpfung von Web-Ressourcen, wie Web-Seiten, Really Simple Syndication (RSS)-Feeds und Web Services, die auf ein einheitliches, REST-basiertes Ressourcenmodell übertragen und dann verknüpft werden. Die Erstellung der Benutzeroberfläche ist jedoch nicht Teil der Komposition – sie erfolgt programmatisch und plattformspezifisch.
Im darauf folgenden Jahr veröffentlichte das Gartner-Institut eine Referenzarchitektur für Enterprise Mashups, die in Abbildung 3.13 zu sehen ist [Bradley, 2007]. Im Zentrum der Schichtenarchitektur steht die *Mashup Assembly* - die Erstellung bzw. Komposition der Mashup-Anwendung. Sie erfolgt durch ein integriertes Entwicklungswerkzeug, welches auf eine Datenzugriffsschicht wie bei der *Mashup Fabric* zugreift und durch die *Processing*-Schicht Unterstützung, beispielsweise bei der Datenmediation, erhält. Die gesamte Erstellung wird durch Infrastrukturkomponenten unterstützt, die beispielsweise die Verwaltung von Komponenten oder die Sicherstellung von QoS übernehmen. Der wachsenden Bedeutung von Community- bzw. Kollaborationsaspekten wird durch eine eigene Schicht Rechnung getragen – hier kann man die gemeinsame Verwaltung, Bewertung und Empfehlung von Mashups und Mashup-Quellen ermöglichen. Die Visualisierung ist explizit von der Mashup-Erstellung getrennt und i. d. R. browserbasiert.

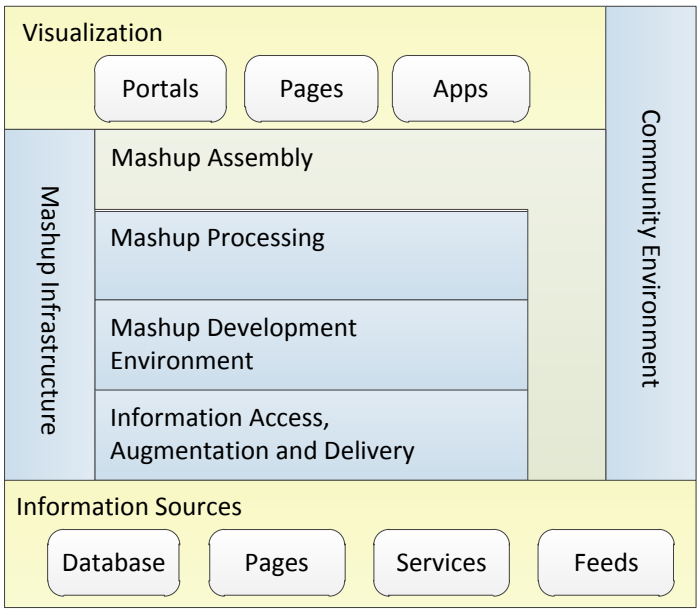

Abb. 3.13: Gartner Referenzarchitektur für Enterprise Mashups [Bradley, 2007]

Auch in diesem Modell ist die Präsentationsebene „aufgesetzt“ und kein Teil der Komposition. Die Autoren empfehlen die Nutzung von Portlets zur Visualisierung, allerdings bleibt unklar, wer diese entwickelt und wie sie in den Kompositionsprozess einbezogen werden. Weitergehende Konzepte für die plattformunabhängige, modellgetriebene Entwicklung oder Adaption sind nicht vorgesehen.

Ähnlich verhält es sich mit dem Mashup-Referenzmodell aus der Sicht von SAP, welches von Hoyer und M. Fischer (2008) vorgestellt wird. Es unterscheidet drei Schichten und Phasen der Entwicklung: *Ressourcen* werden durch Entwickler über einen Uniform Resource Identifier (URI) und Application Programming Interface (API) bereitgestellt. Domänenexperten kombinieren diese Daten bzw. Dienste zu *Widgets* (Anwendungsbausteinen), die später durch Endanwender zu *Mashups* verknüpft werden. López et al. (2008) greifen dieses Modell auf und verfeinern es. Wie Abbildung 3.14 zeigt, wird hier die Erstellung von Daten-Mashups und Widgets explizit getrennt. Für letztere wird analog zum Modell von Bradley (2007) die Portlet-Technologie empfohlen.

Interessant am Modell von López et al. (2008) ist die Möglichkeit, dass Ressourcen visuelles Markup (in Abbildung 3.14 durch die rechte HTML-Ressource verdeutlicht) bereitstellen und als eigenständiges Widget auftreten können. Deren Integration erfolgt auf der Widget-Ebene, was bedeutet, dass eine Kopplung mit andere Daten bzw. Ressourcen nicht möglich ist. Der Mangel eines universellen Kompositionsparadigmas führt zu einer beschränkten Komplexität der Mashup-Lösung, was beim Einsatz als EUD-Werkzeug durchaus gerechtfertigt sein kann, für die Ziele dieser Arbeit aber unzureichend ist. Zur Verknüpfung von Daten-Mashups und Widgets werden verschiedene Alternativen vorgeschlagen. López et al. (2009) zeigen beispielsweise die Aggregation von Daten über das „Data Federation Pattern“, wobei Dienste über relationale Operationen wie JOIN, UNION, SELECTION, usw. miteinander verknüpft werden. Für die Kopplung auf der Präsentationsebene sind die vorgestellten Verfahren nicht ausreichend, da sie sich allein auf Daten konzentrieren und keinerlei

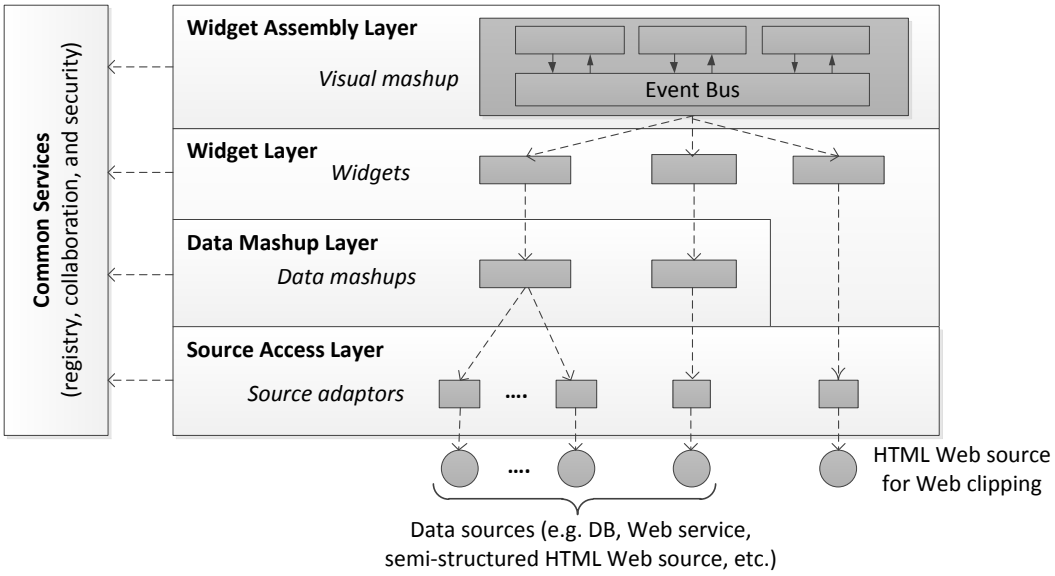

Abb. 3.14: Referenzarchitektur für Enterprise Mashups [López et al., 2008]

Interaktionstechniken abbilden. Modelle und Konzepte zur plattformunabhängigen Entwicklung, zur dynamischen kontextabhängigen Bindung von Diensten oder zur Laufzeitadaption sind durch das Konzept nicht vorgesehen.

Alle genannten Referenzmodelle ermöglichen die Entwicklung interaktiver Anwendungen, die aus verteilten Web-Ressourcen zusammengesetzt werden, durch Nicht-Programmierer bzw. Domänenexperten. Sie sind eher als technische Architekturmodelle zu verstehen – zum Entwicklungsprozess und insbesondere den genutzten Kompositionsmodellen wird keine Aussage getroffen. Im Vordergrund steht die Datenaggregation und -integration, wobei auf ein ressourcenzentrisches Komponentenmodell nach dem RESTful-Prinzip gesetzt wird. Die Präsentationsschicht wird nicht als inhärenter Bestandteil der Komposition verstanden, was die zu Beginn der Arbeit aufgezeigten Probleme bei der UI-Entwicklung unterstreicht.
Verwandte Ansätze aus der Forschung und Technik adressieren häufig nur bestimmte Schichten bzw. Forschungsaspekte der Referenzmodelle, wie Datenextraktion und -analyse, die Werkzeugunterstützung oder die Sicherstellung von Datenschutz und -sicherheit. Die folgenden Abschnitte widmen sich insbesondere Lösungsansätzen zur plattformunabhängigen Modellierung von Kompositionen, der dazu nötigen Infrastrukturunterstützung und Konzepten zur Adaption.

3.2.2.2 Statische Mashups

Die wissenschaftlichen Konzepte zur Entwicklung statischer Mashups reichen von DSLs und visuellen Sprachen bis zu komplexen, semantischen Kompositionsmodellen und -plattformen. Die wichtigsten Vertreter werden im Folgenden vorgestellt und im Hinblick auf die Ziele und Herausforderungen dieser Arbeit bewertet.

DSLs und visuelle Komposition

Ein großer Bereich der Forschung im Mashup-Umfeld beschäftigt sich mit DSLs und visuellen Sprachen zur möglichst einfachen Beschreibung kompositer Anwendungen.

DSLs im Mashup-Umfeld dienen der deklarativen oder programmatischen, syntaktischen Verknüpfung von Web-Ressourcen. Mittels einer natürlichen Syntax auf höherer Abstraktionsebene sollen sie die Entwicklung beschleunigen und es auch Domänenexperten erlauben, Mashups zu erstellen. In *Swashup* [Maximilien et al., 2008] wird hierfür ein Derivat von Ruby-on-Rails [@*Ruby on Rails* 2011] genutzt, welches Sprachkonzepte für die Deklaration von Datentypen, APIs und Diensten besitzt, die über *mashups* und *recipes* verknüpft und gesammelt werden können.
Neben der programmatischen Variante gibt es deklarative Ansätze, wie die Enterprise Mashup Markup Language (EMML) [@OMA, 2009] der *Open Mashup Alliance* – einer Allianz vieler gewichtiger Organisationen mit eigenen Portfolio an Mashup-Lösungen. Die EMML bietet ein XML-Vokabular zur Kopplung von Diensten über relationale Operationen, Gruppierungen und Sortierung von Daten. Es können Bedingungen wie `if/then/else` und Kontrollflusspunkte zur parallelen Abarbeitung von Diensten eingebracht werden. Auch die Integration von Skripten ist möglich.
All jene Vertreter bieten plattformunabhängige Kompositionsmodelle zur Kopplung verteilter Ressourcen. Entsprechend der vorgestellten Referenzmodelle finden sie sich auf der Ebene der Daten-Mashups wieder, da sich die Komposition auf Daten und Geschäftslogik beschränkt. Die Integration oder Kopplung mit UI-Elementen wird nicht unterstützt. Auch Aspekte der dynamischen Bindung oder Adaption der Kompositionen werden nicht betrachtet.

Visuelle Sprachen erlauben die Beschreibung des Datenflusses zwischen Diensten bzw. APIs durch die Verknüpfung ihrer Ein- und Ausgänge und ggf. durch die Angabe von Datentransformationen und -filtern dazwischen. Fast alle populären und bereits genannten Mashup-Plattformen setzen derartige visuelle Werkzeuge ein. Auch in der Forschung widmen sich Projekte wie Damia [Simmen et al., 2008] oder *Next Generation Search* der Thematik. Abbildung 3.15 zeigt die typische Art der visuellen Repräsentation im letztgenanntem Projekt.

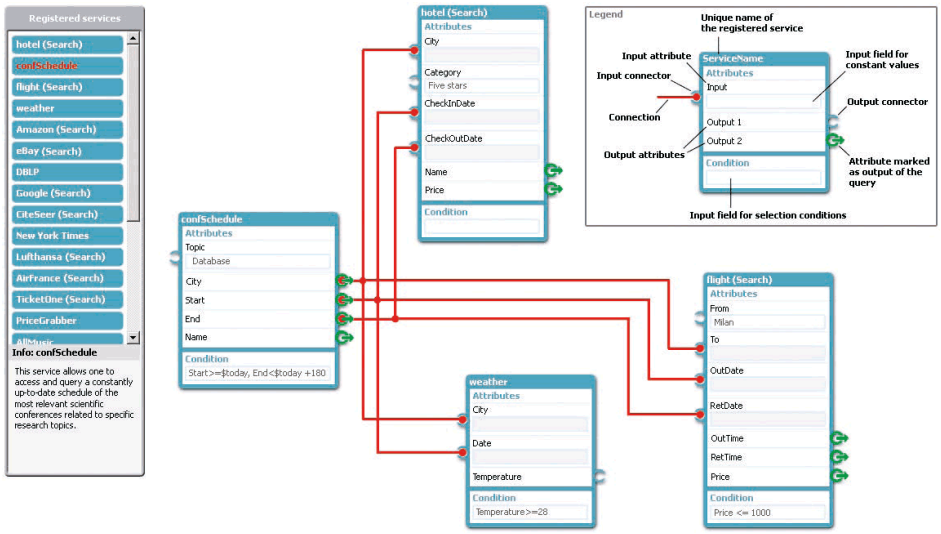

Abb. 3.15: Visuelle Komposition von Service-Mashups [Braga et al., 2008]

Die Mächtigkeit der Vertreter dieser Gattung beschränkt sich auf die visuelle Erstellung von Queries und die Verknüpfung registrierter Dienste mittels Drag-and-Drop. Sie sind somit gut für die Zielgruppe der Nicht-Programmierer geeignet.

Bezüglich der o. g. Referenzmodelle sind sie auf der Daten-Mashup-Schicht verankert und implizieren die sequentielle Ausführung der Dienste im Sinne von Workflows. Die Bereitstellung erfolgt wiederum als Dienst, als Feed oder teilweise durch die Anzeige der Daten auf einer Karte. Die weitere Gestaltung und Komposition der Mashup-Oberfläche, Aspekte der späten Bindung oder Adaptivität werden auch von den Vertretern visueller Mashup-Werkzeuge nicht unterstützt.

Die Komposition auf der Präsentationsebene fand in der Forschung unter den angestrebten Randbedingungen (Plattformunabhängigkeit, Verteilung, lose Kopplung, etc.) bislang wenig Beachtung. Auf die wichtigsten Vertreter und ihre Komponenten- und Kompositionsmodelle wird im Folgenden eingegangen.

Frühe Vertreter zur Web- und UI-Komposition

Viele Ansätze für die komponentenbasierte Entwicklung von Webanwendungen, wie die **XML Application Components** (XAC) [Wiecha et al., 2006], unterstellen die deklarative Spezifikation der Web-Oberflächen. Im Fall von XAC erfolgt die Definition als Kombination von eXtensible HyperText Markup Language (XHTML), XForms und dahinter liegenden Web Services, die über die *State Chart XML Language* miteinander verbunden werden. Grundlage für die Verknüpfung bildet ein gemeinsames XForms Datenmodell. Mit Blick auf die gewünschten RIA-Funktionalitäten stößt der deklarative Ansatz allerdings an seine Grenzen. Außerdem kann die Nutzung einheitlicher Markups und Datenmodelle für Mashups nicht unterstellt werden.

Liu et al. (2007) stellen ein Konzept vor, welches Endnutzer bei der Service-Komposition unterstützen soll. Jede Mashup-Komponente ist hier ein geschlossenes *Modul*, welches aus einer *Service*-, einer *UI*-, und einer *Action*-Komponente besteht – entsprechend dem Model View Controller (MVC)-Muster. Die UI-Komponente verbirgt alle Service-Aspekte hinter einer HTML-basierten Oberfläche. Die Autoren beschränken sich jedoch auf die Beschreibung des rein technischen Konzeptes – Komponenten- und Kompositionsmodelle sowie Konzepte zur nicht-programmatischen Verknüpfung oder späten Bindung von Komponenten werden nicht vorgestellt. Zudem ist festzustellen, dass in dieser Form keine Entkopplung von UI und zugrunde liegenden Diensten verfolgt werden kann, wie es Ziel dieser Arbeit ist.

Diese Entkopplung kann mit **Mashlets** [Abiteboul et al., 2008] erreicht werden, die sowohl auf der Daten-Ebene (*Data Mashlet*) als auch auf der Präsentationsebene (*UI Mashlet*) angesiedelt sein können. Die Verknüpfung mehrerer Mashlets erfolgt wie beim vorherigen Konzept programmatisch und syntaktisch auf der Datenebene. Weitergehende Probleme der UI-Komposition, wie die Anordnung oder Hierarchisierung der UI, werden von den Autoren ebenso wenig adressiert, wie die Verteilung oder späte Bindung von Komponenten.

Im Gegensatz dazu wird in **CompoWeb** [Guo et al., 2008] von Komponenten im klassischen Sinne (vgl. Abschnitt 2.1.1) ausgegangen. *Gadgets* sind lose gekoppelte, komponierbare Anwendungsbestandteile, die zur Laufzeit allein auf Basis ihrer Schnittstellenverträge gebunden werden. Letztere werden mittels XML deklarativ beschrieben. Auf den Entwicklungsprozess und zugrunde liegende Modelle wird nicht näher eingegangen. Stattdessen konzentrieren sich die Autoren auf die isolierte Ausführung von Gadgets in getrennten *Scopes* vor dem Hintergrund der Datensicherheit und Robustheit. Als Lösungsansatz dienen eigene HTML-Tags, die

durch eine entsprechende Browser-Erweiterung verarbeitet werden. Aufgrund dieser *Closed-Corpus*-Beschränkung ist der Ansatz für Mashups nicht geeignet.

WebComposition von Gellersen und Gaedke (1999) stellt zwar den ältesten Kompositionsansatz dar, bietet aber hinsichtlich der Anforderungen die größte Flexibilität. Die Beschreibung einer Webanwendung erfolgt nach dem Konzept deklarativ auf Basis der WebComposition Markup Language (WCML). Allerdings wird nicht die Weboberfläche selbst, sondern Komponenten beschrieben, die hierarchisch geschachtelt werden können und ihrer Gesamtheit die komposite Anwendung darstellen. Komponenten verfügen über einen eigenen Zustand, der durch die Belegung von *Eigenschaften* definiert wird. Auf der Blattebene handelt es sich bei Komponenten um Primitive, wie Text, Tabellen, Bilder, usw., während sie auf höheren Hierarchieebenen ganze Webseiten oder Seiten-Verbünde repräsentieren. Insofern können mit der WCML komponentenbasierte Webanwendungen plattformunabhängig beschrieben werden, wie es das Ziel dieser Arbeit ist.
Das Konzept besitzt jedoch einige Nachteile: Durch die Hierarchisierung verbleibt die Kompositionslogik hinsichtlich des Datenflusses, Layouts, etc. innerhalb der Komponenten bzw. Implementierungsartefakte. Eine unabhängige Modellierung dieser Aspekte wird nicht unterstützt. Im Gegensatz zur verteilten Ausführung und Wartung der aggregierten Web-Ressourcen in Mashups geht der Ansatz zudem von einem zentralen *WebComposition Repository* [Gaedke et al., 1999b] aus, welches den Zugriff und die Ausführung von Komponenten steuert. Das zurückgegebene Markup muss in jedem Fall kompatibel und integrierbar sein – implizit wird von einer homogenen technologischen Plattform ausgegangen. Die Abstraktion von Komponenten im Sinne der dynamischen, plattformabhängigen Auswahl der eigentlichen Implementierung ist somit nicht möglich. Es können jedoch WCML-Deklarationen für verschiedene Plattformen erstellt werden [Gaedke et al., 1999a].
Personalisierungsmechanismen für WebComposition werden von Graef und Gaedke (2000) vorgestellt: Durch *Services* erfolgt die Abstraktion von konkreten Implementierungen, die Aspekte wie Layout, Navigation, Sprache und Inhalte repräsentieren können und deren Auswahl in Abhängigkeit vom Nutzerprofil erfolgt. Der dafür entworfene Algorithmus beachtet allerdings nur die bisherige Verwendung der Dienste durch den Nutzer selbst oder ähnliche Nutzer. Weitergehende Kontextinformationen gehen nicht in die Entscheidung ein. Auch die Anpassung einer bestehenden Komposition und der Austausch von Diensten stehen nicht im Fokus der Betrachtung.

In aller Kürze ist festzustellen, dass alle frühen Kompositions- und Mashup-Ansätze hinsichtlich der Ziele der Arbeit unzureichend sind. Die Komposition auf der Präsentationsebene wird zwar teilweise sogar entsprechend der SOA-Prinzipien unterstützt, allerdings erfolgt die Verknüpfung der Bestandteile rein syntaktisch, programmatisch und auf Basis plattformspezifischer Modelle. Konzepte zur expliziten Modellierung von UI-Aspekten (Layout, Sichten, Interaktivität) oder die späte Bindung und Adaption von Komponenten finden sich in keinem der Ansätze.

Aktuelle Vertreter zur Mashup-Komposition

Einige Forschungsarbeiten aus der jüngeren Vergangenheit adressieren die o. g. Beschränkungen und widmen sich der Komposition unter Berücksichtigung der speziellen Charakteristika von Mashups.

Im Projekt **Mixup** [Yu et al., 2007] wird das Ziel der *Presentation Integration* auf Basis eines zustandsbasierten Komponentenmodells verfolgt. Der innere Zustand von UI-Komponenten wird wie in WebComposition durch Eigenschaften repräsentiert, die durch den Aufruf von *Operationen* verändert werden können, und deren Änderungen über *Ereignisse* nach außen kommuniziert werden. Die Kommunikation und Koordination innerhalb der Anwendung erfolgt datenflussorientiert allein auf Basis dieser Ereignisse. Eine Middleware integriert und verdrahtet die Komponenten mittels Publish/Subscribe. Durch die Nutzung von Adaptern können sie in verschiedenen Technologien, wie JavaScript und .NET, realisiert sein. Die Komposition beschränkt sich jedoch auf die Präsentationsebene - eine explizite Modellierung von Geschäftslogik- bzw. Service-Komponenten ist nicht vorgesehen, was die flexible Kopplung von Backend und Frontend verhindert.

Das Adaptionskonzept für Mixup wird von Daniel und Matera (2008) näher erläutert: *Context Components* dienen der impliziten Bereitstellung von Kontextdaten, z. B. des aktuellen Ortes des Nutzers, als Teil der Komposition. Diese Kontextsensoren sind für die Entwickler – die Endanwender – aber nicht von den sonstigen Komponenten unterscheidbar, da kein Wissen hinsichtlich der Modellierung von Kontext oder Adaptivität vorausgesetzt werden kann. Die damit erreichbare Adaption ist folglich auf die Anpassung auf Schnittstellenebene, wie den kontextspezifischen Aufruf von Operationen, beschränkt. Höherwertige Adaptionen auf Kompositionsebene, wie der Austausch von Komponenten, Änderungen des Layouts oder der Kommunikationsbeziehungen, sind nicht umsetzbar. Zudem kann keine zentrale Konsolidierung, Validierung und Weiterverarbeitung von Kontextinformationen erfolgen, und Konfliktbehandlungen bei widersprüchlichen Adaptionen sind aufgrund der Vermischung mit der Anwendungs- bzw. Kompositionslogik nicht möglich.

Im Nachfolger-Projekt **MashArt** [Daniel et al., 2009] wird der Schritt zur *universellen Komposition* vollzogen, wie sie Ziel dieser Arbeit ist. Abbildung 3.16 zeigt, dass die Beschreibung jeglicher Komponenten ähnlich wie bei Mixup deklarativ, durch die mashArt Description Language (MDL), erfolgt. Diese trennt zwischen Schnittstellenbeschreibung und implementierungsspezifischen *Bindings*. Letztere geben den Typ der Komponente an (*http*, *soap*, *rss*, etc.) und erlauben die Anbindung der Komponente durch spezifische Adapter der Laufzeitumgebung. Mit der Universal Composition Language (UCL) kann die Verknüpfung zwischen Komponenten bzw. deren Ereignissen und Operationen beschrieben werden, wobei der Datenfluss durch die Angabe von XPath-Ausdrücken gefiltert werden kann. Architektonisch erweitert MashArt die clientseitige Mixup-Umgebung um einen serverseitigen Teil, über den ebenfalls entfernte Dienste angebunden werden können.

MashArt kommt den angestrebten Zielen recht nahe, bietet jedoch nur technische Konzepte und weist hinsichtlich der Modellierung der Komposition Schwächen auf: Da die Anwendung zur Laufzeit durch Endnutzer mittels eines Browserwerkzeugs entwickelt wird, ist die Mächtigkeit des Kompositionsmodells bewusst auf die Formalisierung des Datenflusses zwischen Komponenten beschränkt. Die plattformunabhängige Modellierung von Layouts und verschiedenen Sichten ist nicht möglich. Wie bei Mixup beschränkt sich das Konzept auf die Nutzung von HTML-Templates für Layouts, in die Komponenten „eingesetzt“ werden. Zudem sind keine Mechanismen zur kontextadaptiven Bindung von Komponenten oder zur Spezifikation und Umsetzung von Laufzeitadaptivität vorgesehen.

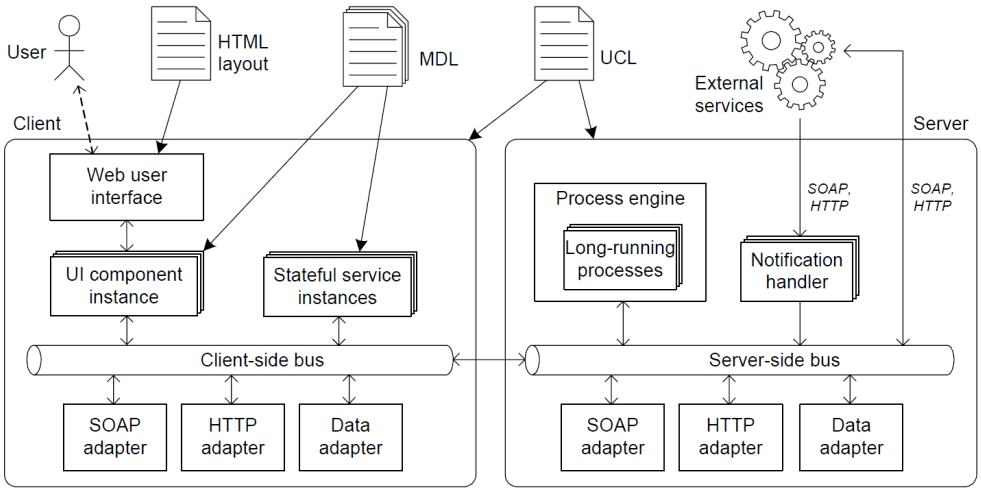

Abb. 3.16: Überblick über das MashArt-System [Daniel et al., 2009]

Das **OMELETTE**-Projekt [@OMELETTE, 2011] widmet sich der Mashup-Entwicklung mit dem Schwerpunkt sog. *Telco Mashups*. Diese erlauben die Verknüpfung von Telekommunikationsdiensten zum Datentransfer oder zur Audio/Video-Kommunikation durch die Endnutzer selbst. Ein wesentlicher Teil des Projektes adressiert die Unterstützung des Autorenprozesses im Sinne des EUD, z. B. durch die Nutzung von Kompositionswissen zur Empfehlung neuer Komponenten [Chowdhury et al., 2011], und ist somit komplementär bzw. ergänzend zur vorliegenden Arbeit zu sehen. Der Ansatz für eine entsprechende Referenzarchitektur, wie von Chudnovsky et al. (2011) skizziert, ist für die vorliegende Arbeit nur bedingt relevant, da auch hier die bereits thematisierte Trennung von Mashups der Daten- und Präsentationsebene besteht. Eine Flexibilisierung durch die lose Kopplung von UI- und Nicht-UI-Komponenten als Teil einer universellen Komposition wird nicht angestrebt. Da die situative Komposition durch Endnutzer im Vordergrund steht, existieren keine Konzepte zur späten Bindung von Komponenten oder zur dynamischen Adaption.

Das EU FP7 Projekt **FAST** [Fernández et al., 2009] verfolgt ebenfalls das Ziel der visuellen Dienstkomposition. Hinsichtlich der Komponentenmodelle und Kompositionsparadigmen liegt das klassische Konzept zustandsloser Dienste zugrunde: SOAP und RESTful Services, aber auch *Forms* mit Benutzerschnittstelle werden als Ressourcen mit definierten Ein- und Ausgängen (*Pre-* und *Post-Condition*) behandelt. Im Gegensatz zu den bislang vorgestellten Systemen sind diese Schnittstellen jedoch semantisch typisiert und die Kopplung erfolgt über *Semantic Pipes*. So können syntaktische Differenzen in Anlehnung an die Mechanismen der SWS bis zu einem gewissen Grad semantisch überbrückt werden. Diesem positiven Aspekt stehen die Zustandslosigkeit der Komponenten, fehlende Modellierungsmöglichkeiten für UI-Aspekte (Interaktion, Layout) und der Mangel an Kontextsensitivität gegenüber. Stattdessen liegt der Schwerpunkt auf dem EUD-Werkzeug.

Wie die semantische Modellierung von Anwendungskomponenten bzw. *Plug-ins* ihre Interoperabilität trotz verschiedener Datenmodelle steigern kann, zeigen Paulheim und Probst (2010). Jedes *Plug-in* wird auf Basis einer Plug-in-Ontologie beschrieben, welches Konzepte aus Interaktions- und Domänenmodellen bereitstellt und miteinander verknüpft. So können Beschreibungen der folgenden Form definiert werden

(Konzepte sind kursiv): „Wenn ein Nutzer eine *Auswahl-Aktion* auf einem *Informationsobjekt* ausführt, welches ein *Hotel* darstellt, so führt die *Kartenkomponente* die *Anzeige-Aktion* auf diesem *Informationsobjekt* aus." Dadurch wird die implizite Verknüpfung von Plug-ins über ihre Interaktionen auf Informationsobjekten möglich. Jedes Plug-in kann aber über ein eigenes Datenmodell verfügen, deren Klassen und Attribute per Annotation an die gemeinsamen semantischen Konzepte gebunden werden. Die Mediation über die semantische Ebene erfolgt transparent durch die Laufzeitumgebung.
Das Konzept verdeutlicht die Möglichkeiten der semantischen Kopplung von Mashup-Komponenten. Darüber hinaus fehlt es jedoch an einem entsprechenden universellen Komponenten- und Kompositionsmodell – die vorgeschlagene Lösung beschränkt sich allein auf die Verknüpfung zustandsloser Java-basierter Komponenten und den Aspekt der semantischen Mediation.

Kommerzielle Mashup-Plattformen

Für die Entwicklung komplexer Enterprise-Mashup-Lösungen existieren ebenfalls eine Reihe kommerzieller Angebote bzw. Plattformen. Sie folgen im Aufbau dem Referenzmodell von Gartner [Bradley, 2007] (vgl. Abschnitt 3.2.2.1) und bieten dedizierte Werkzeuge zur Unterstützung der einzelnen Schichten, insbesondere für die Kopplung von Daten-Diensten mittels visueller Datenfluss-Sprachen (s. o.). Auch die Erstellung von Benutzerschnittstellen für die geschaffenen Daten-Mashups wird von einigen Plattformen unterstützt.
In JackBe Presto [@JackBe, 2011] können z. B. vorgefertigte, plattformspezifische *Widgets* genutzt werden, um Daten in Diagrammen oder Formularen darzustellen. Darüber hinaus müssen die interaktiven *Mashlets* allerdings manuell erstellt werden. Ähnliches gilt für ARIS MashZone [@ARIS, 2011] und die Corizon Mashup Platform [@Corizon, 2011]. Die Kopplung von Daten und UI-Komponenten ist in jedem Fall eng und statisch, und erfolgt auf sehr niedrigem Abstraktionsniveau, z. B. durch explizite Bindung von Datenstrukturen an bestimmte Achsen eines Diagramms.
Die Komponenten- und Kompositionsmodelle der Plattformen bleiben i. d. R. verborgen und die Entwicklung ist somit auf die Lösung des jeweiligen Anbieters beschränkt. Eine Ausnahme bildet das IBM Mashup Center [@IBM, 2011b], welches Komponenten nach der öffentlich zugänglichen iWidget-Spezifikation [@IBM, 2008] komponiert. Die Kompositionslogik beschränkt sich jedoch auch hier auf die einfache, syntaktische, unidirektionale Verbindung von Widgets über Events und verhindert komplexere Koordinations- und Interaktionsmuster, wie sie bereits mehrfach genannt wurden. Weiterhin fehlt es allen kommerziellen Plattformen – wie der Mehrzahl der bereits genannten Forschungsansätze – an Konzepten zur späten Bindung und zur Nutzung von Kontextinformationen zur Kompositions- und Laufzeit.

Fazit

Zusammenfassend kann festgehalten werden, dass aus dem Bereich statischer Mashups für eine Reihe der gestellten Anforderungen vielversprechende Konzepte existieren. Mit *CompoWeb* wurde das Prinzip der lose gekoppelten Software-Komponenten erfolgreich auf das Web übertragen, und durch WebComposition und später Mixup/MashArt mit einem zustandsbasierten Komponentenmodell untersetzt. Durch die Nutzung semantischer Beschreibungen wie in *FAST* wurde ein Weg zur Steigerung der Interoperabilität aufgezeigt.

Die vorgestellten Arbeiten besitzen jedoch Nachteile: Neben der Modellierung des Datenflusses vermag es kein Ansatz, interaktive Mashups vollständig in einem plattformunabhängigen Modell nach dem Vorbild der Web-Engineering-Ansätze (Abschnitt 3.2.1) zu repräsentieren. Die Kommunikation zwischen Komponenten beschränkt sich auf die unidirektionale Weiterleitung von Daten oder Ereignissen – UI-spezifische Anforderungen, wie die Synchronisation von Komponenten oder die permanente Aktualisierung (*Data Refresh*), werden nicht unterstützt. Ferner mangelt es an Lösungen zur dynamischen Bindung und Einbeziehung von Kontextinformationen in den Kompositions- und Ausführungsprozess.
Der Aspekt der Laufzeitadaptivität findet bei der Modellierung von Mashup-Anwendungen bislang kaum Beachtung. Lediglich im Mixup-Projekt wird die Integration von Kontextsensoren in das Anwendungsmodell betrachtet. Das Konzept ist jedoch rein impliziter Natur und äußerst beschränkt (s. o.). Im TELAR-System [Brodt et al., 2008] von Nokia liegt der Fokus auf den technischen Aspekten zur Unterstützung von Kontextadaptivität auf mobilen Endgeräten. Entwicklern wird das *Context Provisioning Framework* an die Hand gegeben, welches Sensordaten der Geräte über das DOM Event-Modell bereitstellt, wozu die Delivery Context: Client Interfaces (DCCI)-Spezifikation [@Waters et al., 2010] genutzt wird. Die Kontextdaten sind somit Name-Wert-Paare und unterliegen keiner Kontextmodellierung und Konsistenzsicherung. Grundsätzlich wird zudem eine programmatische Mashup-Entwicklung mittels HTML und JavaScript unterstellt.
Letztlich bietet kein Ansatz für komposite oder Mashup-Anwendungen ein generisches Modell zur Spezifikation adaptiven Laufzeitverhaltens, Konzepte zur Kontextmodellierung und -verwaltung oder explizite Unterstützung von Adaptionstechniken. Die Umsetzung von Adaptivität bleibt – wenn überhaupt – dem Programmierer der Komponenten oder ggf. der Gesamtanwendung überlassen.

3.2.2.3 Semi-dynamische und dynamische Mashups

Semi-dynamische und dynamische Mashups nutzen i. d. R. semantische Beschreibungen zur dynamischen Bindung oder zur Empfehlung von Komponenten im Entwicklungsprozess oder zur Laufzeit. Die eingesetzten Technologien und Algorithmen bedienen sich der Lösungen, wie sie im SWS-Umfeld etabliert wurden (Abschnitt 3.1.2). Allerdings beschränkt sich die Forschung in diesem Bereich auf einige wenige Ansätze.
So schlagen Ngu et al. (2010) die semantische Beschreibung von Mashup-Komponenten in Form von Swing, Servlets, Widgets oder Portlets mittels SAWSDL vor. Darauf basierend wird ein hybrider Discovery-Algorithmus genutzt, um Anwendungsentwicklern ähnliche Komponenten zu bereits ausgewählten anzubieten. Bianchini et al. (2010) legen bei der Annotation mit SAWSDL ein zustandsbasiertes Komponentenmodell ähnlich dem von MashArt zu Grunde. Semantische Referenzen verknüpfen hier Operationen und Ereignisse von Komponenten mit Konzepten aus Domänenontologien. Die Verwaltung dieser Beschreibungen in einem Komponenten-Repository ermöglicht die Erstellung von *similarity* und *coupling links* zwischen Komponenten. Diese sind durch Koeffizienten quantifiziert, die ein Maß für die semantische Ähnlichkeit und „Passgenauigkeit“ (bzgl. komplementärer Schnittstellen) zweier Komponenten darstellen. Die Links werden zur Empfehlung von

Mashup-Komponenten zur Entwicklungszeit genutzt. Die tatsächliche Integration der Komponenten, die u. U. Anpassungen und Transformationen bei nicht-exakten Übereinstimmungen nötig macht, muss bei beiden Ansätzen durch die Entwickler programmatisch erfolgen. Auch die Modellierung der Komposition, die Nutzung des Mechanismus zur dynamischen Suche und Integration oder zur Datenmediation werden von den Autoren nicht thematisiert.
Gilles et al. (2009) widmen sich der Empfehlung und Integration von *Widgets* zur Laufzeit. Grundlage bietet der *Context Browsing Service*, der permanent über den Zustand einer Anwendung informiert wird und diesen in Form semantischer Fakten speichert, aus denen Situationen geschlussfolgert werden können. So werden Zusammenhänge zwischen Nutzern, Situationen und genutzten Widgets klar. Bei einer Anfrage durch die Ausführungsumgebung – die Autoren nutzen den SAP Rooftop Marketplace – kann das System auf Basis der Fakten Empfehlungen für andere Nutzer aussprechen. Auf die Interna des Dienstes, z. B. hinsichtlich der Ermittlung von Situationen, Empfehlungen, und auf die Mechanismen zur dynamischen Integration und Kopplung der Widgets wird nicht eingegangen.
T. Fischer et al. (2009) gehen einen Schritt weiter und machen die Empfehlung von Komponenten und Inhalten von den vorhandenen Inhalten einer Anwendung abhängig. Der vorgeschlagene Ansatz nutzt dafür das Text-Analyse-Framework UIMA [@UIMA, 2011], welches annotierte Inhalte einer Portal-Anwendung zur Laufzeit analysiert und daraufhin kompatible, mittels OWL-S beschriebene Dienste sucht. Die Beschreibung, welche zusätzlichen Inhalte in welcher Form präsentiert werden sollen, erfolgt durch das *Personalization Model* (ein Overlay-Modell des Domänenmodells der Inhalte). Letztlich entspricht das Konzept einem klassischen Recommender-System, welches sich auf die Anreicherung und Aggregation von Daten bzw. Daten-Diensten beschränkt und der bereits thematisierten Closed-Corpus-Annahme [Brusilovsky, 2001] unterliegt. Hinsichtlich der Integration und impliziten Verdrahtung neuer Komponenten mit der bestehenden Anwendung werden keine Konzepte vorgestellt. Auch die Bereitstellung der entsprechenden Benutzerschnittstelle wird nicht diskutiert.

Alle Ansätze zur dynamischen Komposition von Mashups gehen von semantisch annotierten Komponenten aus und wenden dazu etablierte Konzepte der SWS an. Die Suche nach Komponenten ist sowohl funktional als auch kontextsensitiv möglich, jedoch werden im letzteren Fall die Kontextmodellierung und die Kopplung bzw. Integration mit der Anwendung häufig ausgeblendet.
Der Einsatz der Konzepte beschränkt sich im Normalfall auf die semantische Suche zur Unterstützung von Entwicklern, denen i. d. R. die Auswahl und Integration der Komponenten überlassen wird. Die verwendeten Annotationen bzw. semantischen Konzepte sind an den Status Quo der Web-Service-Welt angelehnt und dienen vorrangig der Typisierung von Ein- und Ausgabedaten. Die Annotation mit Task- und Interaktionssemantik spielt keine Rolle. Auch Konzepte zur dynamischen Bindung oder Adaption auf Basis der semantischen Beschreibungen finden keine Anwendung.

3.3 Zusammenfassung und Diskussion der Defizite existierender Ansätze

Das vorliegende Kapitel widmete sich der Betrachtung des Standes der Forschung und Technik zur Entwicklung interaktiver, komponenten- und dienstbasierter Webanwendungen. Dazu wurden Ansätze aus den Bereichen der Dienstkomposition und SOA sowie des Web- und Mashup-Engineering vorgestellt und analysiert.
Die folgenden Abschnitte fassen die Erkenntnisse aus der Analyse zusammen und stellen den Bezug zu den in Abschnitt 2.3 definierten Anforderungen her, die jeweils mit ✪ markiert sind.

3.3.1 Probleme und Defizite aus dem Bereich der Dienstkomposition

In Abschnitt 3.1 wurden zunächst Lösungskonzepte aus dem Bereich der Dienstkompositionen mit dem Schwerpunkt der Modellierung betrachtet. Es wurde deutlich, dass diese wesentliche Anforderungen der Arbeit adressieren, jedoch nicht ohne weiteres auf die angestrebte universelle Komposition abbildbar sind.
Ausgangspunkt der Betrachtung bildeten wissenschaftliche Konzepte zur **statischen Dienstkomposition**. Deren Modellierung erfolgt i. d. R. auf Basis plattformunabhängiger, formalisierter Modelle bzw. Orchestrierungssprachen wie BPEL. Die ✪ Beschreibung von Diensten erfolgt ebenso formal und technologieunabhängig nach den Prinzipien der ✪ Komponentenorientierung, und auch die ✪ lose Kopplung zu einer Anwendung entspricht den angestrebten Eigenschaften. Für die Modellierung interaktiver Anwendungen sind die Ansätze jedoch ungeeignet, da sie zustandslose Komponentenmodelle und eine rein prozessorientierte Sicht auf die Anwendung propagieren. Die Abbildung von Belangen der ✪ Präsentation und Interaktion ist über derartige Modelle nicht möglich, was der geforderten ✪ Universalität entgegensteht. Zwar existieren mit BPEL4People und WS-HumanTask zwei Spezifikationen zur Integration menschlicher Aktivitäten, diese widmen sich jedoch nicht der visuellen Repräsentation und ermöglichen keinerlei Integration mehrerer UIs. Die gleichen Beschränkungen gelten für ressourcenzentrierte Modelle und Sprachen zur Kopplung von RESTful Web Services.

Danach wurden Lösungen zur **dynamischen Dienstauswahl und -komposition** betrachtet. Auf die Konzepte zur ✪ Discovery und ✪ Mediation im Kontext von SWS wurde ein besonderer Fokus gelegt, da sie starke Anknüpfungspunkte für den angestrebten, kontextsensitiven Kompositionsprozess bieten. Dabei zeigte sich, dass für Teilprobleme, wie die Suche und Auswahl von Komponenten sowie die Mediation von Daten, auf bestehende Erfahrungen und Konzepte zurückgegriffen werden kann.
Als grundlegende Voraussetzung für die ✪ späte Bindung und Komposition wurde die ✪ Abstraktion der einzubindenden Dienste auf Basis von ✪ Semantik identifiziert, wofür zwei alternative Herangehensweisen vorgestellt wurden. Als vielversprechender im Sinne der Arbeit erwiesen sich die leicht verständlichen, gut erweiterbaren und flexiblen annotationsbasierten Konzepte, wie das standardisierte SAWSDL. Sie erlauben die Erweiterung bestehender Beschreibungsformen mit semantischen Referenzen, die gleichsam für interaktive zustandsbasierte Mashup-Komponenten zum Einsatz kommen kann.

Ein Kernziel dieser Arbeit stellt die dynamische, kontextsensitive Suche und Integration von Mashup-Komponenten zur einer interaktiven Anwendung zur Entwicklungs- und Laufzeit dar. Deshalb wurden ✪ Discovery-Algorithmen der SWS vorgestellt, die die Suche von Dienstimplementierungen, ausgehend von den genannten semantischen Beschreibungen, ermöglichen. Beim semantischen Matchmaking auf Basis von Beschreibungslogik werden Suchanfrage bzw. Dienstabstraktion und Angebot bzw. Dienstimplementierung semantisch über Ontologiekonzepte repräsentiert, deren Übereinstimmungsgrad durch Auswertung der Subsumptionsbeziehungen ermittelt wird. Aufgrund ihrer Mächtigkeit, Generik und Entscheidbarkeit können diese Konzepte auch zur Auffindung von Mashup-Komponenten zur Laufzeit genutzt werden. Hierzu bedarf es allerdings Erweiterungen, um der ✪ Universalität bzw. dem Zustand von Komponenten Rechnung zu tragen.
Die Untersuchung von Ansätzen zur Rangfolgebildung zeigte, dass für die dynamische Dienstauswahl bereits ausgereifte Konzepte existieren, die semantisch beschriebene, nicht-funktionale Eigenschaften in den Auswahlprozess einbeziehen und somit dessen ✪ Kontextsensitivität ermöglichen. Die dabei verwendeten Algorithmen sind überwiegend generischer Natur und könnten mit leichten Anpassungen auch auf semantische Beschreibungen von Mashup-Komponenten angewendet werden. Erweiterungen sind u. a. bei den Wichtungskriterien nötig, sodass sie beispielsweise UI-spezifische Annotationen hinsichtlich Interaktionskonzepten und Anforderungen an die Laufzeitumgebung unterstützen. Da im Gegensatz zu Web-Service-Kompositionen im angestrebten Konzept die Komposition nicht von Instanzdaten ausgeht, müssen Matching und Ranking zudem vorrangig auf Kontextparameter Bezug nehmen.
Als weiterer Schritt des SWS-Nutzungsprozesses wurden Lösungen der rekursiven Komposition vorgestellt. Die vollautomatische Erstellung von Ausführungsplänen ist jedoch nicht auf interaktive Mashups anwendbar, da sie einen direkten, funktionalen Zusammenhang zwischen Ein- und Ausgängen von Diensten unterstellt, der bei zustandsbasierten Komponenten mit Nutzerinteraktion nicht gegeben ist. Weiterhin beschränkt sich die Verknüpfung von Web Services auf die Datenebene, während eine rekursive Komposition auf der Präsentationsebene auch Konzepte hinsichtlich des Layouts und verschiedener Sichten bieten muss.
Die Nutzung semantischer Modelle zur Beschreibung von SWS bietet neben der ✪ Abstraktion den Vorteil, syntaktische Inkompatibilitäten überbrücken zu können, was gleichsam für Mashups eine zentrale Herausforderung darstellt. Deshalb wurde exemplarisch die semantische ✪ Mediation über SAWSDL vorgestellt, bei der eine „Brückenontologie“ zum Einsatz kommt. Sie ermöglicht es, zwischen unterschiedlichen strukturellen Repräsentationen bzw. syntaktischen Abweichungen in den Datenmodellen verteilter Dienste zu vermitteln. Konzeptionell ist die Nutzung einer solchen ontologiebasierten Zwischenrepräsentation zur Steigerung der ✪ Interoperabilität auch für Mashup-Kompositionen denkbar, sofern einheitliche Groundings für die annotierten semantischen Datentypen existieren.

Lösungskonzepte zur ✪ Kontextualisierung bzw. **Anpassung von Service-Kompositionen zur Laufzeit** folgen dem MAPE-Modell und nutzen zur Formulierung der Adaptionslogik Wissen, welches durch Funktionen, Regeln oder Policies ausgedrückt wird. Gerade die regelbasierte Beschreibung der Adaptionslogik und die generischen Adaptionsarchitekturen der betrachteten Systeme liefern Anhaltspunkte für die spätere Konzeption. Die Adaptionstechniken beschränken sich jedoch naturgemäß

auf die Daten- und Geschäftslogikebene und umfassen die Beeinflussung von Nachrichten und den Austausch von Dienstimplementierungen. Auch wenn einige Konzepte explizit den Zustand von Komponenten beachten, so sind all diese Lösungen für die Adaptivität in interaktiven Mashups unzureichend, da sie für die Adaptionsmöglichkeiten und -techniken auf der Präsentationsebene keine Modellierungs- und Laufzeitkonzepte bieten.
Abschließend wurden Konzepte zur **Interaktion mit Dienstkompositionen** im Sinne der SOA-Paradigmen vorgestellt. Darunter wies das ServFace-Projekt die größten Stärken hinsichtlich der gestellten Anforderungen auf. Den darin genutzten UI-Annotationen zur ✪ Präsentation von Diensten fehlt allerdings die nötige Mächtigkeit, um dynamisch bedarfsgerechte Benutzerschnittstellen für eine serviceorientierte Anwendung bereitzustellen. Die ✪ lose Kopplung von UI und Diensten im Sinne einer universellen Komposition ist nicht möglich. Vielmehr dient die generierte Oberfläche als Mittel zur visuellen Dienstkomposition – sie ist kein eigenständiger Teil. Ebenso wenig sind Konzepte für die ✪ späte Bindung oder ✪ Kontextualisierung vorgesehen. Ähnliches gilt für die anderen, in diesem Zusammenhang betrachteten Ansätze, die in jedem Fall die manuelle UI-Entwicklung vorsehen. Als visionär ist das SOAUI-Konzept einzuschätzen, welches bereits frühzeitig die Bereitstellung von UI-Bestandteilen in Dienstform propagierte. Den angestrebten Zielen wird es trotzdem nicht gerecht, da eine homogene technologische Basis und deklarative Spezifikation von Komponenten unterstellt wird, was der geforderten ✪ Plattformunabhängigkeit widerspricht. Außerdem wird allein die Komposition der UI betrachtet, womit die Möglichkeit der Entkopplung von Backend und Frontend entfällt.

3.3.2 Probleme und Defizite bei der Entwicklung interaktiver Web- und Mashup-Anwendungen

In Abschnitt 3.2 lag der Schwerpunkt der Betrachtung auf Modellierungsansätzen und Systemarchitekturen für interaktive Webanwendungen und Mashups. Die Analyse zeigte, dass die konzeptionellen Grundlagen traditioneller Web-Engineering-Ansätze nur schlecht mit den Charakteristika kompositer Mashup-Anwendungen vereinbar sind. Die Mehrzahl existierender Mashup-Konzepte stellt sich hingegen zwar als leichtgewichtiges Pendant zur Dienstkomposition dar, bietet aber keine Konzepte zur adäquaten Unterstützung der Präsentationsebene.
Die **Entwicklungsmodelle des Web Engineerings** besitzen aufgrund der langjährigen Entwicklung und vieler Erweiterungen eine hohe Reife. Hinsichtlich der Formalisierung und ✪ Plattformunabhängigkeit können sie als Vorbild für die Konzeption dienen. Die Vorstellung von WebML und UWE als prominenteste und mächtigste Vertreter zeigte jedoch, dass sie aufgrund des starken Bezugs zu Hypermedia-Konzepten, wie Seiten und Links, für die Modellierung von Mashup-Anwendungen nur bedingt geeignet sind. Bis auf einige Erweiterungen zur Einbindung von Web Services finden die Prinzipien der ✪ Komponentenorientierung in den Lösungen keine Anwendung. Schwerpunkt der Systeme ist die Präsentation von und Interaktion mit Daten, nicht die universelle Komposition verteilter Daten und Funktionalitäten. Dabei wird von einheitlichen, vordefinierten Datenmodellen ausgegangen, deren Instanzdaten in bestimmten Formaten vorliegen oder annotiert sein müssen (Closed-Corpus-Problem). Für die ✪ Präsentation von Daten kommen deklarative

Beschreibungen und Transformationsvorschriften zum Einsatz, die die Generierung von HTML-Oberflächen erlauben. Eine ✪ lose Kopplung von Komponenten ist somit – insbesondere auf der Präsentationsebene – nicht möglich. Nicht zuletzt aufgrund dieser Beschränkungen finden sich keine Lösungen zur semantischen ✪ Abstraktion oder für die ✪ späte Bindung von Anwendungsbestandteilen, wenn man von der dynamischen Auswahl aus vordefinierten Varianten absieht.
Als äußerst vielversprechend hinsichtlich der ✪ Kontextualisierung stellt sich die aspektorientierte Modellierung von Adaptionslogik dar, wie sie beispielsweise in UWE verfolgt wird. Aufgrund ihrer Mächtigkeit, Flexibilität und der Entkopplung von der Anwendung kann dieses Prinzip in das zu entwickelnde Konzept einfließen. Die Übertragbarkeit der damit verbundenen, ausgereiften Adaptionsmethoden und -techniken auf Mashup-Komponenten ist allerdings nicht immer gegeben, da die Anpassung auf niedrigem Abstraktionsniveau und implementierungsnah erfolgt. Anpassungen abseits der UI, z. B. für den adaptiven Daten- und Kontrollfluss, wird von der Mehrzahl der Systeme nicht betrachtet. Deshalb wurden komponentenbasierte Ansätze, wie von Nilsson et al. (2006), untersucht, die deutliche Parallelen zu den Mechanismen dienstbasierter Lösungen aufweisen, z. B. hinsichtlich der ✪ Abstraktion von Komponenten und dem Einsatz von Nutzenfunktionen. Im Regelfall beschränken sie sich allerdings auf die Angabe alternativer Implementierungen und die Betrachtung rein syntaktischer Kompatibilität beim Austausch.

Bei der Betrachtung relevanter Arbeiten im Mashup-Umfeld erfolgte zunächst eine Einordnung der angestrebten Ziele anhand verschiedener Klassifikationsschemata und Referenzmodelle in das Forschungsgebiet. Es zeigte sich, dass Mashups aufgrund ihrer Nähe zu Dienstkompositionen in gleicher Weise hinsichtlich ihrer Dynamik und Komplexität kategorisiert werden können. Den Schwerpunkt der folgenden Analyse bildeten Konzepte für die strukturierte, plattformunabhängige Entwicklung statischer Mashups sowie für die Einbeziehung von Kontexteigenschaften in deren Komposition und Ausführung.
Bereits die Untersuchung der existierenden **Mashup-Referenzmodelle** zeigte den Mangel an ✪ Universalität, da Benutzerschnittstellen bislang nicht als inhärenter Teil von Mashup-Kompositionen betrachtet werden. Die ✪ Komponentenorientierung bezieht sich somit nur auf zustandslose Ressourcen, die über Datenflusskonstrukte verknüpft werden. Zur ✪ Präsentation und Interaktion wird häufig, z. B. in den Modellen von Gartner und SAP, die Entwicklung von Portaloberflächen empfohlen, die programmatisch und plattformspezifisch erfolgt. Eine Ausnahme bildet das Modell von López et al. (2008), in welchem zumindest visuelles Markup wie HTML einen Teil der Komposition darstellen kann. Vollwertige, funktionale UI-Komponenten, deren gegenseitige ✪ Koordination und Komposition, z. B. im Sinne von Kontrollfluss, Layout und Sichten, sind allerdings nicht vorgesehen.
Im Weiteren wurden konkrete Mashup-Konzepte aus Forschung und Technik im Hinblick auf die gestellten Anforderungen untersucht. Wie bei Dienstkompositionen erfolgte zunächst die Betrachtung von Lösungsansätzen zur Entwicklung und Bereitstellung **statischer Mashups**. Deren überwiegende Mehrheit widmet sich der Entwicklung visueller oder deklarativer Sprachen und entsprechender Werkzeuge zur Verknüpfung von Inhalten und Funktionalitäten. Die vorgeschlagenen Komponentenmodelle sind zwar i. d. R. plattformunabhängig und einheitlich, entsprechen aber denen der SOA, d. h. sie repräsentieren konventionelle Dienste oder Ressourcen

(RESTful Services). Für die Integration und Kopplung von Benutzerschnittstellen sind sie folglich nicht geeignet.

Einige Forschungsarbeiten beziehen die ✪Präsentation jedoch explizit in die Komposition ein. Davon kommen bereits ältere Vertreter, wie WebComposition, den angestrebten Zielen nahe: Sie ermöglichen die ✪lose Kopplung potentiell verteilter Web-Komponenten, gehen dabei allerdings von homogenen Ausführungsplattformen und Datenmodellen sowie der programmatischen Entwicklung aus.

Neuere Ansätze der *Presentation Integration* stellen Nicht-Programmierer bei der Entwicklung in den Vordergrund. Sie erlauben die Komposition von Anwendungen aus *Widgets*, beschränken sich dabei aber vollständig auf die Präsentationsebene. Die freie Kombination von Daten, Geschäftslogik und Visualisierungsformen – egal ob durch Nutzer, Entwickler oder automatisch – wird dadurch verhindert.

Projekte wie Mixup und MashArt bieten dennoch vielversprechende, universelle Komponenten- und Kommunikationsmodelle, die Anwendungsbestandteile und ihren Zustand einheitlich repräsentieren und die gewünschte ✪Abstraktion von der Implementierungsebene bieten. Diese und vergleichbare Ansätze sehen die datenflussorientierten Kopplung der Bestandteile vor, die sich auf die unidirektionale Weitergabe von Daten – ganz in Tradition der prozessorientierten Sicht der SOA – beschränkt. Reichhaltigere Formen der ✪Koordination, wie die asynchrone Kommunikation, die Synchronisation zwischen Komponenten oder die Koordination durch den Nutzer (Drag-and-Drop), sind in bestehenden Konzepten nicht vorgesehen.

Zur Steigerung der ✪Interoperabilität bei der Verdrahtung setzen einige Systeme, wie FAST, die semantische Typisierung der Komponentenschnittstellen nach dem Vorbild von SAWSDL ein. Die Nutzung von ✪Semantik bildet allerdings die Ausnahme und dient lediglich der Unterstützung des Autorenprozesses. Die kontextsensitive ✪späte Bindung von Komponenten ausgehend von diesen Abstraktionen sowie Mediationskonzepte für die Laufzeit sind nicht vorgesehen.

Die Modellierung von Kompositionen erfolgt zumindest in einigen Systemen deklarativ und plattformunabhängig, beschränkt sich dann aber auf den o. g. Datenfluss. Den Kompositionssprachen mangelt es an ✪Universalität und Offenheit. So bietet kein Ansatz die Möglichkeit der abstrakten Modellierung von Layouts, Sichten oder sonstigen Belangen der ✪Präsentation. Die strukturierte, modellgetriebene Entwicklung interaktiver Mashup-Anwendungen mit den in Abschnitt 2.1.3 beschriebenen Vorteilen ist nach dem aktuellen Stand von Forschung und Technik somit nicht möglich. Die Entwicklung bleibt auf geschlossene, proprietäre Modelle und Plattformen beschränkt, was die ✪Wiederverwendbarkeit, ✪Interoperabilität und Portabilität der entsprechenden Anwendungen schmälert.

Die Betrachtung zeigte ferner einen deutlichen Mangel an Adaptionskonzepten für interaktive Mashups. Weder klassische QoS-Kriterien noch Kontextinformationen zu Ausführungsumgebung oder Nutzer werden in deren Entwicklung und Ausführung einbezogen. Die Anwendbarkeit von Adaptionstechniken aus dem *Adaptive Hypermedia*-Bereich ist als gering einzuschätzen. Hier stehen dokumentenzentrierte Hypertext-Konzepte den Prinzipien zustandsbasierter Black-Box-Komponenten unvereinbar gegenüber. Die ✪Kontextualisierung von Mashups kann demzufolge nur in beschränktem Maße innerhalb einzelner Komponenten, oder aber im Rahmen der programmatischen Erstellung erfolgen.

Als letzter Teilbereich wurden (semi-)automatische bzw. **dynamische Mashups** untersucht, bei denen Konzepte für die ✪Discovery und ✪späte Bindung aus dem SOA-Umfeld auf Mashup-Systeme Anwendung finden. Gerade vor dem Hintergrund der steigenden Anzahl von Komponenten und APIs gewinnen derartige Lösungsansätze an Bedeutung, da sie die Suche und Integration vereinfachen. Die Unterstützung und Automatisierung der Mashup-Komposition adressiert dabei allerdings schwerpunktmäßig den Entwicklungsprozess.
Die Konzepte orientieren sich an den Lösungen aus dem SWS-Umfeld. Hinsichtlich der ✪Abstraktion von Mashup-Komponenten wird von verschiedenen Autoren die semantische Annotation nach dem Vorbild der SAWSDL vorgeschlagen. Auch alle vergleichbaren Ansätze stützen sich auf die ✪Semantik von Komponenten. Sowohl die ✪Beschreibung als auch die ✪Discovery beschränken sich jedoch auf rein funktionale Kriterien, d. h. den Vergleich von Ein- und Ausgaben. Die Einbeziehung nicht-funktionaler und Kontextinformationen wird ausgeblendet.
Insgesamt zeigte sich, dass die Nutzung semantischer Modelle in CBSE und zur UI-Integration in Kinderschuhen steckt. Maße, wie die semantische Ähnlichkeit und Passgenauigkeit zwischen Komponenten, werden zumeist zur Empfehlung von Komponenten im Autorenprozess genutzt. Die ✪Interoperabilität auf syntaktischer Ebene muss dann i. d. R. programmatisch durch Entwickler sichergestellt werden. Die wenigen Systeme, die die Empfehlung zur Laufzeit ermöglichen, gehen indes von gemeinsamen Datenmodellen aus und sind stark an aktuelle Recommender-Systeme angelegt. Wie die tatsächliche, dynamische Integration von Komponenten zur Laufzeit erfolgt, wird nicht erläutert.

Die geschilderten Defizite unterstreichen deutlich die Herausforderungen, die in Kapitel 2 dargestellt wurden. Es zeigt sich, dass die bestehenden Forschungsarbeiten den identifizierten Anforderungen in ihrer Gänze nicht gerecht werden, aber vielversprechende Teillösungen für die spätere Konzeption bieten. So empfiehlt sich die Übertragung und Erweiterung der SOA-Konzepte zur semantischen Annotation, funktionalen Suche und Mediation auf Mashups. Ebenso können einige Komponenten- und Anwendungsmodelle aus dem Web-Engineering-Umfeld als Grundlage für die Konzeption dienen, da sie eher auf die Interaktivität der Anwendungen und Präsentationsaspekte zugeschnitten sind.
Keines der vorgestellten Konzepte und Systeme wird jedoch allen Anforderungen gerecht. Eine ganzheitliche Lösung, die die plattformunabhängige Entwicklung und Beschreibung interaktiver Webanwendungen aus universellen, zustandsbehafteten Komponenten erlaubt, und deren dynamische kontextsensitive Komposition unterstützt, muss deshalb erst geschaffen werden.
Zur Lösung der skizzierten Probleme wird in den folgenden Kapiteln ein entsprechendes Konzept zur modellgetriebenen Entwicklung adaptiver Mashup-Anwendungen aus den Blickwinkeln der Modellierung, des Kompositionsprozesses und der Systemarchitektur vorgestellt. Das folgende Kapitel gibt zunächst einen Überblick über das Gesamtkonzept.

4

Universelle Komposition adaptiver Webanwendungen – Modellgetriebene Entwicklung und Systemstützung

Zur Lösung der in den letzten Kapiteln identifizierten Probleme und Defizite wird in dieser Arbeit ein neuer, modellbasierter Entwicklungsansatz für komposite Mashup-Anwendungen vorgeschlagen. Gegenüber bisherigen Lösungen hebt er sich durch die einheitliche Nutzung serviceorientierter Paradigmen auf allen Anwendungsebenen ab. Das Kompositionsprinzip für funktionale und Datendienste in bestehenden Dienst- und Mashup-Plattformen wird damit auf die Präsentationsschicht erweitert und ermöglicht die von Daniel et al. (2009) vorgeschlagene *universelle Komposition* von Webanwendungen. Letztere bestehen folglich aus Komponenten, die heterogene, verteilte Ressourcen aller Anwendungsebenen repräsentieren und einem einheitlichen Komponentenmodell unterliegen.

Dieses Konzept ermöglicht erstmals die strukturierte Entwicklung universeller Kompositionen auf Basis eines plattformunabhängigen Modells. Es obliegt den entsprechenden Laufzeitumgebungen, dieses Modell zu interpretieren, wenn es nicht bereits durch vorherige Transformationsschritte in eine ausführbare Anwendung umgewandelt wurde. Eine Schlüsselrolle bei der Ausführung nimmt die dynamische Suche und Integration von Komponenten ein, die auf Basis semantischer Repräsentationen nach dem Vorbild von SWS und unter Einbeziehung von Kontext- und nichtfunktionalen Eigenschaften erfolgt. Neben dieser Adaption zur Kompositionszeit unterstützt das Konzept explizit die kontextabhängige Anpassung von Anwendungen zur Laufzeit, wofür sowohl im Kompositionsmodell als auch in der Laufzeitumgebung entsprechende Mittel vorgesehen sind.

Dieses Kapitel gibt einen Überblick über das Gesamtkonzept und stellt, ausgehend vom zugrunde liegenden Rollenmodell, die wesentlichen Prinzipien des modellgetriebenen Entwicklungsprozesses, der kontextsensitiven Komposition und Integration zur Laufzeit sowie der Adaptionsmechanismen vor. In den nachfolgenden Kapiteln werden diese Konzepte im Detail erläutert.

4.1 Grundkonzept und Rollenmodell

Der universellen Komposition von Webanwendungen liegt ein gegenüber dem klassischen Verständnis von SOA erweitertes Rollenverständnis zugrunde. Abbildung 4.1 zeigt die am Erstellungsprozess eines interaktiven Mashups beteiligten Akteure und Werkzeuge. Dienstanbieter (*Service Developer*) stellen wie bisher domänen- und anwendungsspezifische Daten und Geschäftslogik über etablierte Schnittstellen und Protokolle (*Services*) zur Verfügung. Mit Hilfe existierender Werkzeuge, z. B. Service-Mashup-Plattformen oder BPEL-Engines, können diese bereits jetzt zu kompositen Prozessen bzw. Diensten (*Composite Services*) kombiniert werden.
Durch das neue Konzept werden nun auch Ressourcen auf der Präsentationsebene – Benutzeroberflächen samt domänenunabhängiger Präsentationslogik – als wiederverwendbare Dienste verstanden und modelliert (UIS). Ein universelles Komponentenmodell erlaubt es, die Dienste der Daten-, Geschäftslogik- und Präsentationsebene nach einheitlichen Prinzipien zu repräsentieren und mit Hilfe dedizierter Werkzeuge oder Erweiterungen bestehender Tools zu einer kompositen Anwendung (*Composite Application*) zu verknüpfen. Durch den hohen Abstraktionsgrad des dabei verwendeten Modells und die Werkzeugunterstützung wird es auch Nicht-Programmierern – im Mittelpunkt stehen versierte Nutzer und Fachexperten, die ein grundlegendes Verständnis der fachlichen Modelle und Funktionen besitzen – möglich, ihre konkreten Probleme auf eine Mashup-Komposition abzubilden.

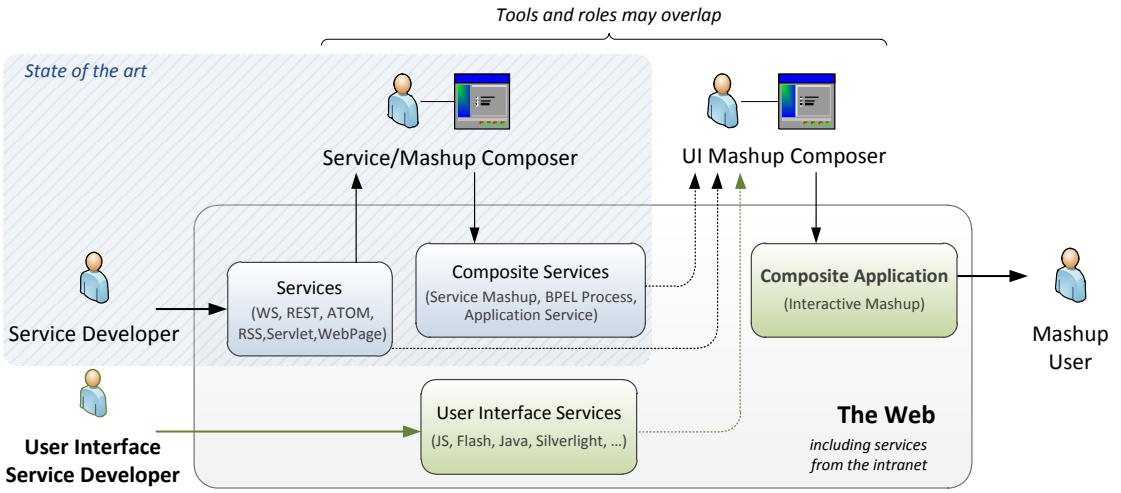

Abb. 4.1: Erweitertes Rollenmodell im Kontext der universellen Komposition

Als beteiligten Rollen ergeben sich folglich:

Service Developer sind Entwickler funktionaler Anwendungsbestandteile, die Daten oder Geschäftslogik über standardisierte Schnittstellen und Protokolle verfügbar machen. Klassischerweise kommen hierbei Web Services auf Basis von SOAP oder REST zum Einsatz. Zur Integration in eine komposite Anwendung müssen diese als Mashup-Komponenten entsprechend des im nächsten Kapitel vorgestellten Modells gekapselt werden.

Service/Mashup Composer verknüpfen funktionale und Datendienste zu komplexeren Diensten oder Prozessen. Dazu können die im letzten Kapitel vorgestellten Kompositionssprachen und Werkzeuge aus dem SOA- und Mashup-Bereich

zum Einsatz kommen. Derartige Kompositionen können i. d. R. wieder über eine Dienst-Schnittstelle angesprochen werden.

UI Service Developer entwickeln Komponenten, die über eine eigene Benutzeroberfläche und möglichst generische Präsentationslogik verfügen. Diese sollte möglichst unabhängig von konkreten Datenmodellen und Anwendungsfällen sein, um die Kopplung mit verschiedenen Diensten zu erlauben. Wie bei konventionellen Diensten setzt die Entwicklung von UIS grundlegende Programmierkenntnisse voraus und folgt dem später vorgestellten Komponentenmodell.

UI Mashup Composer verknüpfen konventionelle und UI-Dienste zu einer interaktiven Anwendung, mit der Endnutzer interagieren können. Diese Komposition hat die Lösung eines konkreten Anwendungsproblems unter Einbeziehung des Nutzers zum Ziel und setzt keine Programmierkenntnisse voraus. Ein Grundverständnis der fachlichen Domäne, der funktionalen Bestandteile und ihrer Verknüpfung ist für die Erstellung jedoch nötig.

Mashup User sind die Endnutzer der kompositen Anwendung und können diese punktuell an ihre Bedürfnisse anpassen. Erweiterte Techniken zur Komposition und Koordination durch Endnutzer (EUD) stehen allerdings nicht im Mittelpunkt dieser Arbeit.

Wie in Abbildung 4.1 zu erkennen ist, kann die Erstellung von Geschäftslogik und Benutzerschnittstellen im Sinne der SoC getrennt erfolgen. Die Integration beider Ebenen kann im Gegensatz zu bestehenden Ansätzen nun auf der Grundlage eines einheitlichen Paradigmas erfolgen. Die kontinuierliche Wartung und Evolution der Dienstinhalte – dies betrifft Daten, Logik und Benutzerschnittstellen gleichermaßen – kann indes unabhängig von Anwendungen erfolgen.
Selbstverständlich können je nach Anwendungskontext einige Rollen der Entwicklung zusammenfallen. So können Web- und UI-Services durch die gleichen Entwickler bereitgestellt und Mashups durch ihre Nutzer selbst erstellt werden. Da eine Komposition trotz aller Vereinfachungen aber ein grundlegendes Verständnis der fachlichen Konzepte und der Verknüpfungslogik voraussetzt, wird davon ausgegangen, dass Fachexperten für die spezifizierten Szenarien zunächst geeignete Komponenten suchen und verbinden, bevor sie Endnutzern zur Verfügung gestellt werden.
Im Folgenden werden die für diese Vision notwendigen wissenschaftlichen Beiträge dieser Arbeit skizziert und im Zusammenhang erklärt. Die konzeptionellen Details der Modellierung und Ausführung derartiger Anwendungen werden in den nachfolgenden Kapiteln 5 und 6 vorgestellt.

4.2 Modellgetriebene Entwicklung kompositer Mashups

Wie im letzten Abschnitt bereits angedeutet wurde, basiert das Konzept der universellen Komposition von Mashup-Anwendungen auf einem einheitlichen Komponentenmodell für all ihre Bestandteile. Dieses Modell abstrahiert von Eigenheiten der jeweiligen Technologien und Implementierungen und erlaubt es, Komponenten aller Anwendungsebenen allein auf Basis ihrer definierten öffentlichen Schnittstellen zu einer interaktiven Anwendung zu koppeln, wie es bislang mit Web Services nur auf der Daten- und Geschäftslogikebene möglich war.

Die folgenden Abschnitte geben einen Überblick über die Modelle zur Repräsentation von Komponenten und deren Verknüpfung zu einer interaktiven Anwendung.

4.2.1 Universelles Komponentenmodell

Grundlage einer Komposition nach den angestrebten Zielen bildet ein universelles Komponenten- und Kommunikationsmodell für Mashup-Anwendungen, welches die Kapselung von Datendiensten, Geschäftslogik und UI-Bestandteilen gleichermaßen in Form interoperabler Komponenten ermöglicht. Die Benutzeroberfläche einer Anwendung ergibt sich somit als Mashup von UI-Komponenten, die auf Geschäftslogik und Daten von darunter liegenden Komponenten zurückgreifen.
Komponenten können beispielsweise Web Services auf Basis von REST und SOAP, Feeds, aber auch JavaScript-APIs und -Widgets repräsentieren. Eine derartige Kapselung kann entweder automatisch (im Falle einfacher Web Services), semi-automatisch (wie von Daniel und Matera (2009) gezeigt) oder manuell erfolgen. Durch eine deklarative Komponentenbeschreibung (MCDL, Abschnitt 5.1.3) können Angaben zur äußeren Schnittstelle und Bindung von Komponenten beschrieben werden – analog zur WSDL in der Web-Service-Domäne. Auf Basis der dadurch verfügbaren Informationen kann die einheitliche, plattformunabhängige Beschreibung, Verwaltung und Integration von Komponenten erfolgen. Insbesondere die Nutzung einer derartigen Beschreibungsform zur dynamischen Bindung und Integration verteilter UI-Ressourcen in Anwendungen stellt ein Novum dar.
Das später genauer vorgestellte Komponentenmodell adressiert somit exakt die im Abschnitt 1.1 vorgestellten Probleme: Es erlaubt die Abstraktion von Implementierungsdetails einer Komponente, wodurch die plattformunabhängige Modellierung und modellgetriebene Entwicklung von kompositen Anwendungen ermöglicht wird. Erstmals ist dies konsequent für die gesamte Anwendung, einschließlich der Präsentationsebene, möglich. Dem enorm hohen Aufwand bei der UI-Entwicklung kann durch die Integration bestehender, ausgereifter UI-Komponenten direkt entgegengewirkt werden. Entwicklung und Wartung aller Anwendungsbestandteile können parallel und unabhängig von den kompositen Anwendungen erfolgen.
Die Nutzung eines universellen Komponenten- und Kommunikationsmodells eröffnet weiterhin die Möglichkeit, Adaptionsstrategien über alle Anwendungsebenen hinweg einheitlich zu formulieren. Gleichermaßen können einzelne Ebenen explizit adaptiert werden, z. B. zur Flexibilisierung der Anwendungsoberfläche, indem UI-Komponenten kontextabhängig ausgewählt, angepasst und ausgetauscht werden, ohne gleichzeitig die dahinter liegende Geschäftslogik und Datenbasis zu verändern.
Auf die konzeptionellen Einzelheiten von Komponentenmodell und -beschreibung wird ausführlich in Abschnitt 5.1 eingegangen.

4.2.2 Belangorientiertes Kompositionsmodell

Die Konzepte des o. g. Komponentenmodells bilden die Grundlage für die Modellierung kompositer Anwendungen, bei der Komponenten zu einer interaktiven Anwendung zusammengesetzt werden. Zur Beschreibung solcher Kompositionen wird in dieser Arbeit ein Kompositionsmodell vorgeschlagen, welches die Verknüpfung von Komponenten auf abstrakter Ebene und unabhängig von der Ausführungsplattform

beschreibt. Es kann als „Pendant“ zu BPEL verstanden werden, dessen Instanzen interaktive Anwendungen hinsichtlich der zu integrierenden Komponenten, deren anwendungsspezifischer Konfiguration, des Layouts, der Kommunikation, sowie der Interaktion und Adaptivität beschreiben. Im Gegensatz zu Modellen der Dienstkomposition wird die Benutzerschnittstelle explizit in die Modellierung einbezogen.

Abbildung 4.2 veranschaulicht die Zusammenhänge zwischen den Modellen. Rechts oben ist das im vorigen Abschnitt angesprochene Komponentenmodell zu sehen, welches zur Typisierung der Komponentenschnittstellen auf Konzepte verschiedener Domänenmodelle zurückgreift. Die Implementierungen der Komponenten (links unten) arbeiten zur Laufzeit indes auf Daten, deren Syntax durch sog. *Groundings* der semantischen Typen vordefiniert sind. Das Kompositionsmodell (links oben) integriert und verknüpft Komponenten allein auf Basis ihres abstrakten Modells. Die vollständigen Kompositionen werden schließlich durch Kompositions- bzw. Laufzeitumgebungen interpretiert, was die kontextsensitive Bindung von Komponentenartefakten erlaubt, die dem Zielmodell entsprechen.

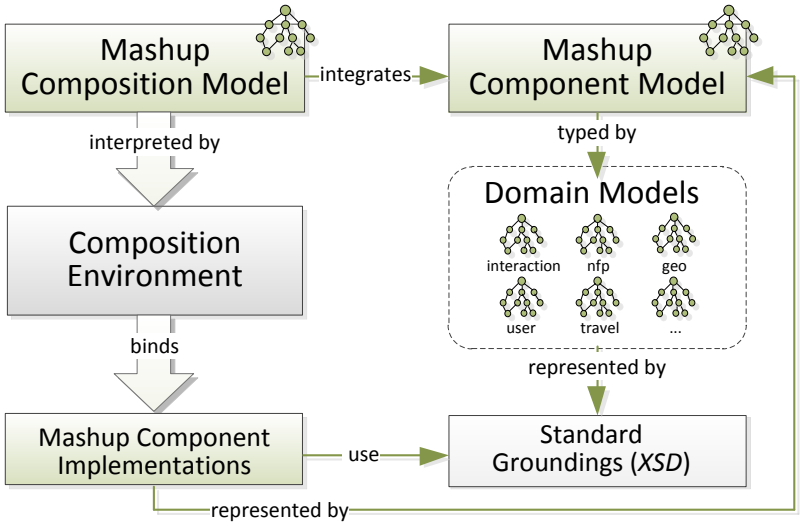

Abb. 4.2: Überblick über die Modellnutzung im Konzept

Die Modellierung von Kompositionen erfolgt belangorientiert, d. h. das Kompositionsmodell zerfällt in Teilmodelle, die verschiedene Belange von Mashup-Anwendungen separat beschreiben.

Alle grundlegenden *Konzepte* einer Mashup-Anwendung – in erster Linie die zu integrierenden Komponenten – werden darin entsprechend dem gemeinsamen Komponentenmodell repräsentiert. Ihre Kopplung erfolgt ereignis- und datenflussorientiert, wobei verschiedene Mechanismen zur Steigerung der Interoperabilität zum Einsatz kommen. Syntaktische Differenzen zwischen den Schnittstellen und Datenmodellen der Komponenten können beispielsweise durch Abbildungsvorschriften im Modell und die semantische Typisierung überbrückt werden. Gegenüber konventionellen Sprachen zur Dienstkomposition stehen im Kompositionsmodell weiterhin Klassen zur Verfügung, die sich explizit der UI-Modellierung widmen. So können beispielsweise verschachtelte Layouts zur Anordnung von UI-Komponenten und verschiedene Sichten definiert werden.

Die Erweiterbarkeit um zusätzliche Anwendungsbelange wird durch ein Teilmodell verdeutlicht, welches Entwicklern die Möglichkeit gibt, Änderungen an einer Komposition in Abhängigkeit von Kontexteigenschaften zu spezifizieren. Die Modellkonzepte erlauben die aspektorientierte Definition von Adaptionen, wie sie in UWE eingeführt wurde (vgl. Abschnitt 3.2.1), und unterstützen sowohl komponentenspezifische (Rekonfiguration, Austausch) als auch komponentenübergreifende Adaptionstechniken (adaptives Layout, adaptiver Datenfluss).

Das Kompositionsmodell setzt die in Abschnitt 2.3 formulierten Anforderungen und Prinzipien, wie die lose Kopplung aller Bestandteile und die Abstraktion von technologischen Eigenheiten der Komponenten und Laufzeitumgebungen, direkt um. Dadurch ergeben sich eine Reihe von Vorteilen, die den problematisierten Herausforderungen entgegenwirken:
Die Modellierung von Anwendungen erfolgt unabhängig von der späteren Ausführungsplattform. Ein fachliches Modell bildet die Grundlage für die Ausführung auf verschiedenen Plattformen. Dazu wird es entweder durch Transformatoren umgewandelt oder durch die Laufzeitumgebungen selbst dynamisch interpretiert. Einzige Voraussetzung für die erfolgreiche Ausführung ist, dass für die Plattform kompatible Komponenten existieren, was ggf. durch Technologieadapter erreicht werden kann.
Die Abstraktionen im Modell sind gleichsam Ausgangspunkt für die Kontextualisierung der Mashups. Die Entscheidung, welche Datenquelle, welcher Algorithmus oder welche Visualisierung in Form einer konkreten Komponentenimplementierung zum Einsatz kommt, kann auf den Initialisierungszeitpunkt der Anwendung verschoben werden. Sie kann nicht nur durch die Gegebenheiten der Laufzeitumgebungen, sondern durch beliebige Kontexteigenschaften beeinflusst werden. Das fachliche Modell der Anwendung bleibt davon unabhängig, sodass es einfach erweitert und verändert werden kann – ohne Berücksichtigung von Implementierungsaspekten. Weiterhin können nicht nur Komponenten wiederverwendet werden, sondern auch Kompositionen, die ihrerseits Komponenten darstellen. Alles in allem sinken die zeitlichen und finanziellen Aufwände der Anwendungserstellung und -wartung.

Das Kompositionsmodell wird ausführlich in Abschnitt 5.2 vorgestellt, die unterstützten Adaptionstechniken und deren Modellierung in Abschnitt 5.3.

4.3 Dynamische Integration und Laufzeitumgebung

Zur Ausführung der modellierten, kompositen Webanwendung bedarf es der Unterstützung durch entsprechende Laufzeitumgebungen und Infrastrukturdienste. In dieser Arbeit wird deshalb eine Plattform vorgestellt, welche den Daten- und Kontrollfluss gemäß dem zugrunde liegenden Kompositionsmodell steuert und die modellierte Adaptivität zur Laufzeit umsetzt. Diese Ausführungsumgebung greift wiederum auf Dienste zurück, die u. a. die Verwaltung von Komponenten und von Kontextdaten anwendungs- und plattformübergreifend erlauben.
Durch ihren kompositen Charakter können Anwendungen in verschiedener Art und Weise an Plattform und Kontext angepasst werden: (1) durch die kontextsensitive Auswahl zu integrierender Komponenten, (2) durch die kontextabhängige Konfiguration dieser Komponenten, (3) durch die nutzer- oder systemgetriebene Anpassung

einer bestehenden Komposition (Rekonfiguration, Austausch von Komponenten, Änderung des Layouts, usw.), sowie (4) durch Komponenten selbst im Rahmen ihr eigenen Adaptivität. Die letztgenannte Form der Anpassung ist durch die Komposition nicht direkt beeinflussbar, da sich die Kenntnis über Komponenten auf deren Schnittstelle beschränkt. Sie kann und muss durch Komponentenentwickler auf Basis bestehender Konzepte umgesetzt werden. Die vorliegende Arbeit konzentriert sich dementsprechend auf die ersten drei Möglichkeiten der Kontextualisierung.
Die Spezifikation der bedarfsgerechten Konfiguration (2) und Laufzeitadaption (3) wird durch das skizzierte Kompositionsmodell erlaubt. Vor der Initialisierung einer Anwendung muss jedoch die kontextabhängige Auswahl der im Modell beschriebenen Komponenten erfolgen, die im nächsten Abschnitt überblicksartig vorgestellt wird. Die Grundkonzepte der adaptiven Laufzeitumgebung werden danach diskutiert.

4.3.1 Kontextsensitiver Integrationsprozess für Mashup-Komponenten

Das gerade vorgestellte Kompositionsmodell bildet den Ausgangspunkt für die Komposition bzw. Integration und Ausführung einer jeden kompositen Anwendung. Die universelle Repräsentation von Komponenten bietet die Chance, Konzepte der dynamischen Dienstauswahl und -anpassung aus dem SOA-Umfeld (Abschnitt 3.1.2) auf alle Bestandteile einer Mashup-Anwendung, einschließlich ihrer Benutzerschnittstelle anzuwenden. Aufgrund ihrer konsequenten Entkopplung können sie dynamisch und kontextabhängig ausgewählt werden. Der damit verbundene Integrationsprozess für Mashup-Komponenten durchläuft verschiedene Phasen, die Abbildung 4.3 zusammenfasst.

Abb. 4.3: Prinzipieller Ablauf der Integration von Mashup-Komponenten

Nach der Modellierung ① wird zunächst das Kompositionsmodell interpretiert ②, wobei insbesondere die Extraktion der abstrakten Komponentenrepräsentationen von Interesse ist. Diese bilden die Grundlage für die nächsten beiden Schritte. Beim *Matching* ③ werden für jede abstrakte Komponente im Modell deren funktionale Anforderungen mit tatsächlich verfügbaren Komponenten abgeglichen. Dabei wird auf die Informationen aus den Komponentenbeschreibungen (Abschnitt 5.1.3) zurückgegriffen. Alle hinsichtlich ihrer Schnittstelle und Funktionalität passenden Komponenten werden im nächsten Schritt der Rangfolgebildung (*Ranking*) ④ bezüglich nicht-funktionaler und kontextabhängiger Eigenschaften bewertet und (aus-) sortiert. Die dazu benötigten Strategien können als Teil des Kompositionsmodells spezifiziert werden. Schließlich wird die Komponente integriert bzw. gebunden, die den Anforderungen am nächsten kommt ⑤.
Der Prozess der kontextsensitiven Integration von Mashup-Komponenten weist Parallelen zum SWS-Nutzungsprozess aus Abbildung 3.5 auf. Ausgangspunkt bildet jedoch nicht eine einzige Dienstanfrage samt Instanzdaten, sondern eine Vielzahl von Zielbeschreibungen, die in Form abstrakter Komponenten im Kompositionsmodell definiert sind. Die Nutzung ähnlicher Konzepte zur Abstraktion und semantischen

Typisierung erlaubt es, Algorithmen und Lösungsansätze aus dem SWS-Bereich zumindest teilweise bei der Integration von Mashup-Komponenten einzusetzen. Ein Beispiel hierfür ist die Mediation ⑥ zwischen semantisch kompatiblen, allerdings syntaktisch inkompatiblen Komponenten zur Laufzeit, die an die Konzepte von SAWSDL angelehnt ist. Sie erlaubt die automatische Umwandlung ausgetauschter Daten über gemeinsam genutzte semantische Repräsentationen (vgl. Abschnitt 3.1.2).

Die Bedeutung des Integrationsprozesses wird mit Blick auf die identifizierten Probleme und Ziele der Arbeit klar: Die kontextsensitive Auswahl von Komponenten zur Initialisierungszeit einer Anwendung erlaubt es, zur Entwicklungszeit von Eigenheiten der Ausführungsplattform und -umgebung zu abstrahieren. Wie bereits beschrieben, führt dies zur Steigerung von Wiederverwendbarkeit, Qualität und Portabilität der geschaffenen Anwendungen und gleichzeitig zur Senkung des zeitlichen und finanziellen Aufwands bei der Entwicklung. Weiterhin können beliebige Kriterien, wie Kontext- und QoS-Eigenschaften in die Auswahl einfließen.
Auf die einzelnen Schritte und Mechanismen des Integrationsprozesses wird ausführlich in Abschnitt 6.1 eingegangen.

4.3.2 Referenzarchitektur zur adaptiven Komposition und Ausführung interaktiver Mashup-Anwendungen

Abbildung 4.4 zeigt die neue Infrastruktur zur Generierung, kontextsensitiven Bereitstellung und Ausführung kompositer Webanwendungen im Überblick. Die Zahlen dienen der Zuordnung von Infrastrukturkomponenten zu den Schritten des eben erläuterten Integrationsprozesses.

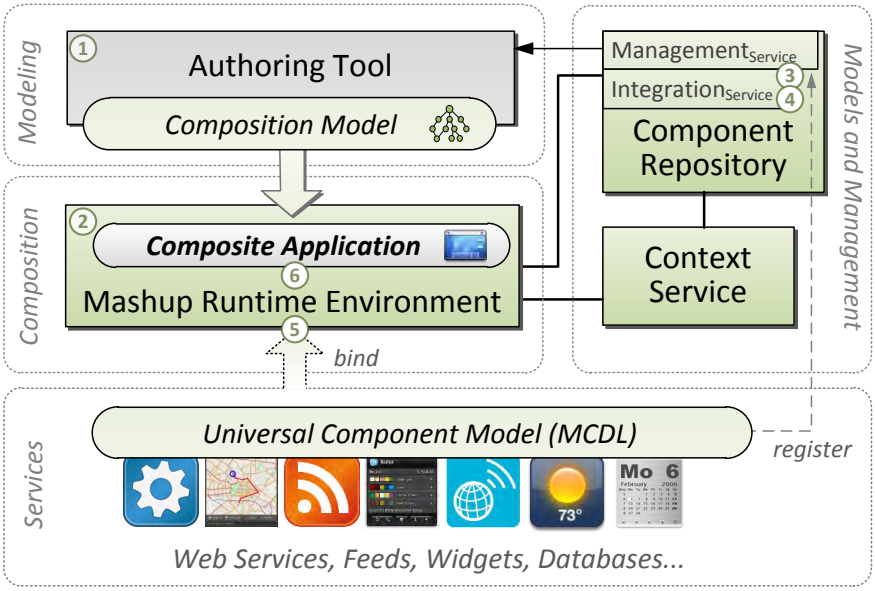

Abb. 4.4: Konzeptionelle Übersicht der verteilten Kompositionsinfrastruktur

Die Architektur integriert die im unteren Bereich angedeuteten Dienste, welche Daten, Anwendungslogik und UI-Bestandteile bereitstellen und innerhalb der Komposition durch das in Abschnitt 5.1 beschriebene Komponentenmodell einheitlich repräsentiert werden. Im rechten Teil der Abbildung sind die Dienste zur

semantischen Verwaltung von Komponenten – in Folge als Component Repository (CoRe) bezeichnet – und Kontextwissen (*Context Service*) zu sehen. Der linke Teil verdeutlicht den modellgetriebenen Charakter des Konzeptes. Ausgangspunkt bildet das plattformunabhängige Kompositionsmodell einer Anwendung, welches mit Hilfe geeigneter Autorenwerkzeuge erstellt wird. Letztere stehen nicht im Fokus dieser Arbeit, auch wenn zur Erleichterung des Autorenprozesses ein beispielhafter Editor prototypisch umgesetzt wurde (vgl. Abschnitt 7.1.4).
Der Zeitpunkt der Verarbeitung des Modells ist konzeptionell offen gehalten und kann auf verschiedene Arten erfolgen: Es kann (1) statisch (zur Designzeit) oder (2) dynamisch (zur Initialisierungszeit) in Anwendungscode transformiert, oder (3) direkt durch Laufzeitumgebungen interpretiert werden.
Die resultierende Anwendung besteht aus Komponenten, die über ihre universellen Schnittstellen – beschrieben durch das angesprochene Komponentenmodell – direkt oder gemäß dem im letzten Abschnitt erläuterten Integrationsprozess gesucht und gebunden werden. Dazu wird auf die Such- und Auswahlmechanismen des *Integration Service* im CoRe zurückgegriffen.
Die Initialisierung und Ausführung einer Anwendung übernimmt eine Laufzeitumgebung, die Mashup Runtime Environment (MRE), deren technologische Basis konzeptionell nicht beschränkt ist. Sie bietet kompositen Mashups eine Plattform, die neben der Integration und Initialisierung in erster Linie die Anwendungsfunktionalität gemäß den obligatorischen Teilen des Kompositionsmodells sicherstellt. Dazu zählen u. a. das Life-Cycle-Management der integrierten Komponenten, die Umsetzung des Kontroll- und Datenflusses auf Basis einer ereignisorientierten Kommunikationsinfrastruktur und die Bereitstellung einer einheitlichen Schnittstelle für den Zugriff auf entfernte Dienste.
Die Verortung von MRE und Anwendung zwischen Client und Server ist konzeptionell freigestellt. Sie kann sowohl rein clientseitig, d. h. entsprechend der *Thin Server Architecture* nach Prasad et al. (2007) angesiedelt sein, oder wird als konventionelle Client-Server-Architektur umgesetzt. Die Entscheidung, welche Plattform zur Ausführung geeigneter ist, muss beim Deployment bestimmt werden. Anhaltspunkte hierfür können die Zielgruppe und ihre Endgeräte sowie die verfügbare serverseitige Infrastruktur, aber auch sicherheitskritische Anforderungen wie der Einsatz im geschäftlichen Rahmen (vgl. *Enterprise Mashups*, Abschnitt 2.1.2) geben.
Wie Abbildung 4.4 zeigt, sind die Belange der Mashup-Ausführung, der Verwaltung, Suche und Bereitstellung der Komponenten und der Kontextverwaltung im Sinne der SoC getrennt. Letztere werden als unabhängige Dienste angeboten und können somit anwendungs- und plattformübergreifend genutzt werden. Diese serviceorientierte Struktur bringt Vorteile, wie die getrennte Wartung und bessere Skalierbarkeit der Kompositionsumgebung, mit sich, da die verschiedenen Dienste ggf. auf mehrere Server verteilt werden können.
Eine genauere Darstellung der für die universelle Komposition benötigten Systeminfrastruktur und Laufzeitumgebung bietet Abschnitt 6.2.

4.3.3 Unterstützung von adaptivem Laufzeitverhalten in Mashups

Bereits der Auswahlprozess für Komponenten greift auf Kontextinformationen zurück und dient somit der Adaptivität. Als Teil des Kompositionsmodells kann

allerdings auch adaptives Laufzeitverhalten einer Mashup-Anwendung definiert werden. Dieses ist i. d. R. an Kontextänderungen gebunden, die überwacht, modelliert und interpretiert werden müssen. Um den hohen Aufwand für die Erfassung und Verwaltung von Kontextwissen von der Anwendung zu trennen, wurde das Konzept eines dedizierten, ontologiebasierten Kontextdienstes entwickelt. Dieser kümmert sich um die Modellierung, Konsistenzsicherung, Validierung und Ableitung spezifischer, höherwertiger Kontextparameter. Seine Offenheit erlaubt die Anbindung beliebiger externer Kontextmonitore und impliziert die anwendungs- und plattformübergreifende Kontextnutzung.
Die MRE tritt ihm gegenüber als Kontextanbieter auf, der z. B. Endgeräteeigenschaften bereitstellt. Gleichzeitig nutzt es das modellierte Kontextwissen selbst, um komposite Anwendungen dynamisch an veränderliche Gegebenheiten anzupassen. Grundlage hierfür bieten die Adaptionsaspekte im Kompositionsmodell. Sie beschreiben Änderungen an Teilen der Anwendung, die bei Kontextänderungen unter bestimmten Bedingungen ausgeführt werden.
Die dabei unterstützten Adaptionstechniken reichen von der Rekonfiguration einzelner Komponenten bis hin zu weitreichenden Anpassungen des Daten- und Kontrollflusses. Eine besondere Herausforderung gegenüber adaptiven Dienstkompositionen stellt der Fakt dar, dass Komponenten über einen Zustand und eine Benutzeroberfläche verfügen können. Die MRE bietet deshalb Konzepte zur Zustandssicherung und zur automatischen Anpassung des Layouts beim dynamischen Komponententausch. Die einheitliche Umsetzung der vielfältigen Adaptionsmöglichkeiten universeller Kompositionen stellt gegenüber existierenden Kompositionsarchitekturen für Dienste, Web- und Mashup-Anwendungen eine wissenschaftliche Neuerung dar. Einzelheiten der Adaptionsarchitektur und ihrer internen Abläufe werden in Abschnitt 6.3 vorgestellt.

Die nächsten beiden Kapitel widmen sich jeweils einem der beiden Blickwinkel dieser Arbeit. In Kapitel 5 liegt der Schwerpunkt der Betrachtungen auf den Konzepten der Entwicklungszeit, d. h. insbesondere die Konzepte zur Modellierung und Beschreibung von Komponenten und Kompositionen werden diskutiert. Kapitel 6 stellt im Anschluss die entwickelten Laufzeitkonzepte für adaptive komposite Mashup-Anwendungen vor. Im Mittelpunkt stehen hier der angesprochene kontextsensitive Integrationsprozess, die skizzierte Referenzarchitektur zur Komposition und Ausführung sowie die Adaptionsmechanismen der Laufzeitumgebung.

5

Belangorientierte Modellierung adaptiver, kompositer Webanwendungen

Im letzten Kapitel wurde ein Überblick über das Gesamtkonzept der modellgetriebenen Entwicklung adaptiver komponentenbasierter Mashups gegeben. Grundlage dafür stellt ein plattformunabhängiges Metamodell dar, welches die belangorientierte Spezifikation eines interaktiven Mashups erlaubt. Dazu stellt es Syntax und Semantik für die Beschreibung aller nötigen Anwendungsbelange bereit, u. a. für die genutzten Komponenten, das Layout sowie den Kontroll- und Datenfluss. Zusätzlich erlaubt es die Spezifikation adaptiven Verhaltens, welches die Anpassung der Komposition an Kontextänderungen zur Laufzeit beschreibt.
Dieses Kapitel widmet sich der Darstellung des Entwicklungsprozesses mit einem Schwerpunkt auf diese Modellierungsmittel. Als Richtlinie für die Konzeption können die von Assmann (2003) angeführten Schlüsselkonzepte für komposite Anwendungen und deren Anforderungen herangezogen werden. Demnach gilt es zunächst, ein **Komponentenmodell** zu entwickeln, welches die einheitliche Repräsentation der Anwendungsbestandteile unter der Beachtung von Modularität, Parametrisierbarkeit sowie standardisierten Schnittstellen erlaubt. Eine passende **Kompositionstechnik** muss Verknüpfungskonzepte bereitstellen, um die möglichst flexible Kopplung der Komponenten unter Berücksichtigung von Heterogenität und Erweiterbarkeit zu ermöglichen. Zuletzt muss eine **Kompositionssprache** entwickelt werden, die Formalismen zur Beschreibung kompositer Anwendungen bereitstellt.
Dementsprechend werden in Abschnitt 5.1 zunächst ein universelles Komponentenmodell für Mashup-Anwendungen sowie die zugehörige Beschreibungssprache vorgestellt. Abschnitt 5.2 geht danach näher auf das belangorientierte Metamodell ein, welches die Kompositionssprache darstellt und die Verknüpfung der beschriebenen Komponenten zu einer interaktiven Mashup-Anwendung erlaubt. Die verschiedenen Anwendungsbelange sind darin in Teilmodelle getrennt, die nacheinander vorgestellt werden. Ein eigenes Unterkapitel ist der Modellierung des adaptiven Laufzeitverhaltens gewidmet. Zuletzt wird das Vorgehen bei der Modellierung illustriert, bevor eine Zusammenfassung und Diskussion das Kapitel abschließen.

5.1 Ein universelles Komponentenmodell für Mashup-Anwendungen

Die universelle Komposition von Anwendungen basiert auf der Idee, dass all ihre Bestandteile auf den verschiedenen Anwendungsebene demselben Komponentenmodell unterliegen. Diese Komponenten – seien es konventionelle Web Services oder Bestandteile der Benutzerschnittstelle – müssen deshalb einheitlich repräsentiert und mit wohldefinierten Schnittstellen versehen werden.
Wie aus der Betrachtung verwandter Ansätze klar geworden ist, lässt sich die Komplexität einer Anwendung nicht beliebig abstrahieren oder gar auflösen, ohne zu trivialen oder sehr beschränkten Lösungen zu führen. Mit dem hier vorgestellten Konzept wird ein Großteil der Komplexität „hinter“ die Komponentenschnittstelle verschoben. Letztere entkoppelt die Spezifikation einer Komponente von Implementierungsdetails gemäß dem *Information-Hiding*-Prinzip analog zu Web Services. Dies bedeutet gleichzeitig, dass Komponentenentwicklern eine sehr wichtige Rolle zukommt, da der Bau wiederverwendbarer Komponenten äußerst aufwändig ist.
Neben der Fehlerfreiheit und guten Dokumentation sollten Komponenten insbesondere testbar sein, z. B. indem bereits Testdatensätze mitgeliefert werden. Um echte Wiederverwendbarkeit gewährleisten zu können, muss Wert auf die sinnvolle Parametrisierung und Nutzung generischer Datenmodelle gelegt werden. Die Nutzung in verschiedenen, vorher unbekannten Anwendungskontexten stellt zudem hohe Ansprüche an die Robustheit und Skalierbarkeit. All diese Herausforderungen kann ein Nicht-Programmierer oder Endanwender nicht bewältigen, weshalb zur Erstellung von Komponenten – ebenso wie von Web Services – erfahrene Programmierer gefragt sind. Die einfache und kostengünstige Entwicklung kompositer Anwendungen durch Nicht-Programmierer macht insofern einen erhöhten Initialaufwand bei der Komponentenentwicklung nötig.
Die Entwicklung von Komponenten steht nicht im Fokus der Arbeit. Sie sollte nach gängigen Vorgehensmodellen, verbunden mit einer umfassenden Anforderungserfassung aus Sicht der *UI Mashup Composer* (vgl. Abbildung 4.1) erfolgen. Im Rahmen studentischer Arbeiten, wie [Kreska, 2008], wurden zudem Konzepte entwickelt, um bestehende Komponenten verschiedenen Typs (Web Services, JavaScript Widgets, Applets, etc.) möglichst automatisiert zu kapseln und als Komponente gemäß dem entwickelten Modell bereitzustellen. Der folgende Abschnitt stellt die damit verbundenen Eigenschaften und Charakteristika näher vor.

5.1.1 Grundlegende Eigenschaften und Prinzipien

Komponenten stellen wiederverwendbare „Bausteine“ einer Anwendung dar, die Daten, Geschäftslogik, Benutzerschnittstellen oder Kombinationen daraus zur Verfügung stellen. Den Prinzipien von Web Services folgend, unterstellt das Modell keine inneren Strukturen oder Formate. Es definiert jedoch fundamentale Charakteristika der Komponentenschnittstellen, welche die Grundlage für ihre spätere Integration und Kommunikation bilden.
Ein Ziel der Arbeit ist die Steigerung der Wiederverwendbarkeit webbasierter Benutzerschnittstellen. Wiederverwendbare Softwareressourcen sind allerdings häufig unbrauchbar, wenn die verwendeten Konzepte dem Verständnis und somit

ihrer erfolgreichen Integration in Softwaresysteme entgegenstehen [Kim und Stohr, 1998]. Deshalb ist das Komponentenmodell einfach und generisch gehalten. Wie in Abbildung 5.1 zu sehen ist, werden Komponenten danach auf einer nicht-technischen Ebene durch drei Abstraktionen beschrieben: *Property*, *Event* und *Operation*.

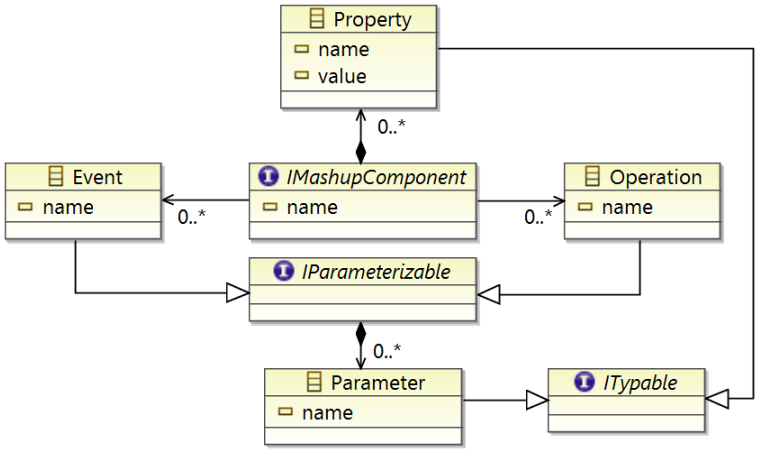

Abb. 5.1: Die Abstraktionen bzw. Klassen des universellen Komponentenmodells

Die Menge der *Properties* einer Komponente repräsentiert ihren (von außen sichtbaren) Zustand, wie er bereits durch Daniel et al. (2009) für interaktive Komponenten motiviert wurde. Er wird über Name-Wert-Paare definiert, die durch den Komponentenentwickler spezifiziert werden müssen und alle für die Umgebung bzw. die Konfiguration relevanten Daten umfassen. Mit Properties verbundene Werte können sowohl primitiver Natur sein, aber auch komplexe Daten enthalten, die beispielsweise mittels XML Schema definiert werden. Beispiele für typische Eigenschaften sind Höhe und Breite von UI-Komponenten, die genutzte Sprache, oder der Modus einer Karte (Normal, Satellit, Hybrid).
Um der Umgebung interne Zustandsänderungen mitzuteilen, publizieren Komponenten *Events* (Ereignisse). Sie können auf verschiedene Art und Weise ausgelöst werden, u. a. durch Nutzerinteraktionen, interne Logik, oder nach Rückmeldung externer Dienste. Neben einem Namen sind Events durch eine beliebige Anzahl an Parametern charakterisiert. Diese verfügen, wie Properties, über einen Namen und Datenwert. Ein typisches Event im Szenario einer Reiseplanung (Abschnitt 2.2.1) würde in Folge der Ortsauswahl durch den Nutzer auf einer Karte ausgelöst, um den ausgewählten Ort zu symbolisieren (*LocationSelected(GLatLong location)*). Neben dem Datenfluss lässt sich über Events der Kontrollfluss einer Anwendung realisieren, da sie nicht zwingend Daten in Form von Parametern transportieren müssen. Vielmehr können auch leere Steuernachrichten publiziert werden, z. B. nach der erfolgreichen Initialisierung oder dem Erreichen eines sicheren Zustands.
Die dritte Abstraktion im Modell stellen *Operationen* dar – Methoden einer Komponente, die durch Events ausgelöst werden. Sie können beliebige Funktionalitäten beinhalten bzw. auslösen, wie interne Zustandsänderungen, Berechnungen oder Dienstaufrufe. Als Eingabe verwenden sie die Parameter der aufrufenden Events. Eine exemplarische Operation *getCountryInfo(GLatLong location)* könnte durch das oben erwähnte Event ausgelöst werden und einen SOAP-Request zur Abfrage von Informationen über das betreffende Land anstoßen. Die daraus resultierenden

Informationen können durch die Komponente selbst dargestellt werden, oder wiederum durch ein Event nach außen verfügbar gemacht werden.
Properties und Parameter sind typisiert (setzen ITypable um). Die Ausprägung der Typisierung ist konzeptionell freigestellt und kann je nach Ausführungsumgebung variieren. XML- und JSON-Schema-Datentypen können ebenso wie Referenzen auf semantische Konzepte angegeben werden. Für das weitere Konzept wird die semantische Typisierung durch Referenzierung von Entitäten aus Ontologien zugrunde gelegt, wobei die Komponenten zur Laufzeit mit XML-Instanzdaten arbeiten. Die entsprechenden Konzepte zur Modellierung sowie Kommunikation und Mediation werden später genauer erörtert.
Abbildung 5.2 zeigt am Beispiel einer Kartenkomponente zur Routenberechnung die Repräsentationsformen entsprechend des vorgestellten Modells (links) und die Ausprägung zur Laufzeit (rechts). In orange sind die zustandsgebenden Properties, in grün die Operationen und in blau die ausgehenden Events zu sehen.

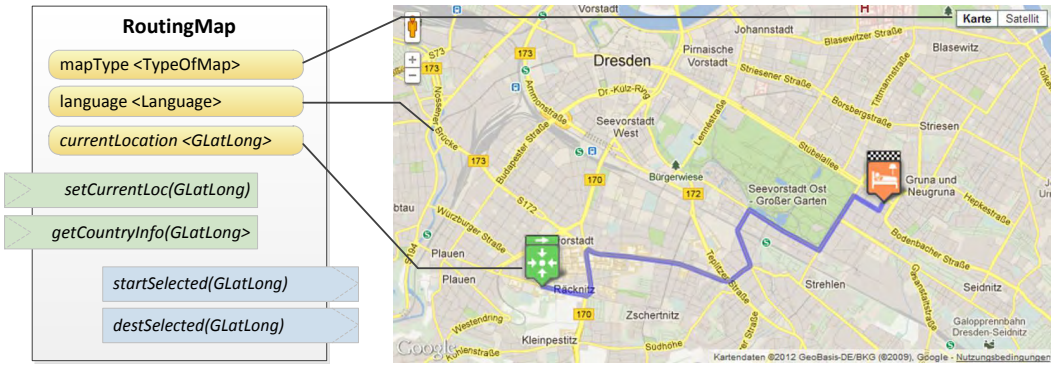

Abb. 5.2: Komponentenrepräsentation im Modell (links) und zur Laufzeit (rechts)

5.1.2 Komponententypen

Dank dem universellen Ansatz lassen sich verschiedenste Bestandteile einer Anwendung als Komponenten repräsentieren. In erster Linie handelt es sich dabei um die Teile, die Kernfunktionalitäten der Anwendung erbringen, wobei zwischen Komponenten mit und ohne Benutzerschnittstelle unterschieden werden kann. Die Unterscheidung hat keinen Einfluss auf die Modellierung, muss jedoch aufgrund der unterschiedlichen Behandlung durch die Laufzeitumgebung getroffen werden. Sie manifestiert sich als Attribut der Komponentenbeschreibung, welches durch Entwickler entsprechend gesetzt werden muss.
UI-Komponenten entsprechen der gängigen Vorstellung von „Widgets“. Sie besitzen im Regelfall eine grafische Oberfläche und erlauben dem Nutzer die Interaktion mit der Anwendung. Dabei ist es aus Sicht der Komposition irrelevant, ob die Komponenten ausschließlich auf der Präsentationsebene angesiedelt sind, ob sie eigene Anwendungslogik besitzen oder auf entfernte Dienste zugreifen. Das prominente Beispiel *Google Maps* zeigt, dass eine solche Trennung nicht immer ohne weiteres möglich ist[1]. Komponenten können ebenso nur Geschäftslogik oder Daten

[1] Die Maps API [@Google, 2011b] erlaubt die Einbindung einer interaktiven Karte in die Anwendung, die zwar in erster Linie rein der Interaktion dient, letztlich aber im Hintergrund auf komplexe Logik zum Laden und Anzeigen neuer Kartenbestandteile zurückgreift.

bereitstellen. Im Kontext dienstbasierter Anwendungen handelt es sich hierbei i. d. R. um gekapselte Web Services - sog. *Service-Komponenten*.
Die Festlegung der Granularität von Komponenten stellt eine Herausforderung in der Entwicklung dar, die insbesondere auf der Präsentationsebene auftritt. Auf den Ebenen darunter ist die Granularität mit der von Web-Services vergleichbar. Im Hinblick auf die Wiederverwendbarkeit und die Zielgruppe des Kompositionsansatzes kann die *Funktionalität aus Endnutzersicht* als Gradmaß herangezogen werden: UI-Komponenten decken eine atomare Funktionalität einer Anwendung ab. Anhand eines Beispiels von Laga et al. (2009) lässt sich dies veranschaulichen. In einer E-Mail-Anwendung haben Nutzer die Möglichkeit, Textnachrichten zu verfassen, die Adresse des Empfängers anzugeben, Dateien anzuhängen, die Nachricht zu senden, den Posteingang zu überwachen, E-Mails zu lesen, und zu antworten. Aus Endnutzersicht bestehen die drei wesentlichen Funktionalitäten im (1) Verfassen und Senden, (2) Überprüfen des Posteingangs, und (3) Lesen einer E-Mail. Folgerichtig bietet sich die Aufteilung in drei UI-Komponenten an. Eine gröbere oder feinere Unterteilung wird vom Kompositionsansatz unterstützt, geht allerdings zu Lasten von Wiederverwendbarkeit oder Komplexität.

Neben den Anwendungsbausteinen können auch alle weiteren „Beteiligten“ an der Komposition als Komponenten modelliert werden. Dies betrifft zunächst die Laufzeitumgebung selbst (vgl. Abschnitt 5.2.1), denn insbesondere bei der Einbettung in eine größere Anwendung, spielt diese Schnittstelle der Komposition „nach außen“ eine tragende Rolle. So wurden im Rahmen des diese Arbeit umgebenden Forschungsprojektes CRUISe [@CRUISe, 2012] komposite Anwendungen in das CRM-Portal CAS PIA [@CAS, 2011] integriert. Für den Datenaustausch mit der Rahmenanwendung wurde letztere ebenfalls als Komponente modelliert, was die Kommunikation mit Anwendungskomponenten über Events ermöglichte.
Weiterhin muss zur Unterstützung von Adaptivität in kompositen Anwendungen eine Verknüpfung mit dem Kontextmodell erfolgen. Letzteres kann ebenfalls als Komponente modelliert werden, deren Eigenschaften den Kontext bzw. spezifische Kontextparameter repräsentieren (vgl. Abschnitt 5.3.2.1). Bei Veränderungen können sie der restlichen Komposition über Ereignisse verfügbar gemacht werden.
Die Repräsentation all dieser Bestandteile als Komponenten erlaubt es, den Daten- und Kontrollfluss, die Kommunikation mit Rahmenanwendungen und Plattform, sowie Kontextabhängigkeiten innerhalb einer Komposition nach einheitlichen Prinzipien zu definieren.

5.1.3 Beschreibung von Komponenten

Komponenten bzw. ihre Schnittstellen müssen einheitlich beschrieben werden, um eine universelle Komposition zu ermöglichen. Zu diesem Zweck wurde in Anlehnung an die WSDL die deklarative, technologieunabhängige Sprache Mashup Component Description Language (MCDL) entwickelt. Sie beschreibt neben den oben genannten funktionalen Schnittstellenbestandteilen (Properties, Operations, Events) nicht-funktionale Eigenschaften von Komponenten sowie Informationen zu deren Bindung, Integration und Ausführung zur Laufzeit. Die folgenden Abschnitte stellen die Designprinzipien und die verschiedenen Ausprägungen der Sprache mit aufsteigender Komplexität vor.

5.1.3.1 Designprinzipien und Überblick

Die identifizierten Anforderungen (Abschnitt 2.3) und zur Abbildung des Komponentenmodells benötigten Konzepte weisen starke Parallelen zur WSDL auf. Im Gegensatz zur Beschreibung konventioneller Web Services adressiert die MCDL jedoch Anwendungsbestandteile, die nicht zwangsläufig zustandslos sind und komplett entfernt ausgeführt werden. Insbesondere bei Komponenten, die auf der Präsentationsebene angesiedelt sind, ist es nötig, sie auf dem Endgerät des Nutzers zu integrieren und auszuführen. Dies macht entsprechende Integrationsmechanismen und Laufzeitumgebungen nötig, bringt aber auch zusätzliche Anforderungen an die Beschreibung mit sich. So müssen insbesondere Anforderungen an die Ausführungsumgebung und Informationen zur Steuerung des Lebenszyklus' einer Komponente für die Laufzeitumgebung ersichtlich sein.

Nach dem Vorbild der WSDL wird deshalb eine deklarative, XML-basierte Sprache vorgeschlagen, die sowohl einfach und menschenlesbar ist als auch schnell und effizient maschinell verarbeitet werden kann. Sie besteht aus einer abstrakten Beschreibung der eigentlichen Schnittstelle entsprechend des vorgestellten Komponentenmodells, und einem *Binding*-Teil, der die Abbildung konkreter Implementierungsartefakte auf die Schnittstelle ermöglicht. Die Modellierung kann demzufolge rein auf Basis der Schnittstelleninformationen erfolgen, während die Binding-Informationen der Suche, Integration und Ausführung einer Komponente dienen. Sie geben u. a. Aufschluss über die technologischen Anforderungen, die Konfigurations- und Steuerungsmöglichkeiten. Zur Unterstützung der Suche wird die Angabe weiterer Metadaten ermöglicht, die in Abschnitt 5.1.3.2 erörtert werden.

Zur Beschreibung von Mashup-Komponenten wird auf Sprachen verschiedener Mächtigkeit zurückgegriffen (vgl. Abbildung 5.3). Die MCDL bildet die Basissprache und stellt das grundlegende Vokabular zur Schnittstellendefinition bereit. Die Semantic MCDL (SMCDL) bietet Mittel zur semantischen Typisierung und Annotation von Komponenten hinsichtlich der Funktionalität und Abhängigkeiten zwischen Ereignissen und Operationen. Die ontologiebasierte Form MCDO nutzt schließlich die volle Mächtigkeit der zur Verfügung stehenden Beschreibungslogik aus, und erlaubt u. a. die Formulierung von Vor- und Nachbedingungen für Operationen und die Gewichtung von Annotationen. Auf die Rolle der *Templates* als Zielbeschreibung von Komponenten im Entwicklungsprozess wird in Abschnitt 5.2 eingegangen.

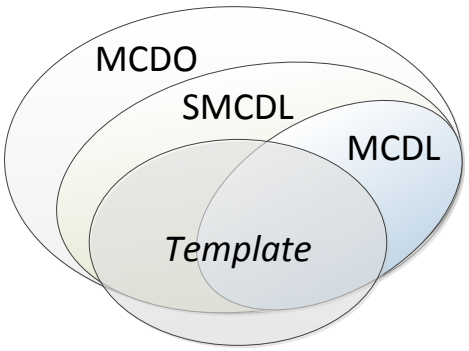

Abb. 5.3: Unterstützte Komplexitätsgrade der Komponentenbeschreibungen

5.1.3.2 Mashup Component Description Language (MCDL)

Die Mashup Component Description Language (MCDL) ist eine XML-basierte, deklarative Beschreibungssprache für Komponenten gemäß dem beschriebenen Modell, die als Erweiterung und Verallgemeinerung der User Interface Service Description Language (UISDL)[2] [Hübsch et al., 2010] entstand. Wie bereits angesprochen, orientiert sie sich an den Designprinzipien und Strukturmitteln der WSDL. Letztere ist in einen abstrakten und konkreten Bereich unterteilt. Ersterer definiert die Schnittstelle eines Dienstes (*interface*) mit den angebotenen Operationen und Datentypen in einer plattform- und implementierungsunabhängigen Form. Der konkrete Teil (*binding*) beschreibt, wie eine spezifische Implementierung bzw. Instanz des Dienstes aufgerufen werden kann.
Die MCDL folgt dieser Unterteilung mit den Elementen `interface` und `binding`. Die Modellierung kann allein auf Basis der Schnittstelleninformationen, und somit unabhängig von bestimmten Technologien oder Plattformen erfolgen. Die Binding-Informationen werden zur Suche, Integration und Ausführung von Komponenten, also spätestens beim Laden der Anwendung, benötigt. Sie geben u. a. Aufschluss über die technologischen Anforderungen, die Konfigurations- und Steuerungsmöglichkeiten. Zur präziseren Suche ist die Angabe weiterer Metadaten möglich.

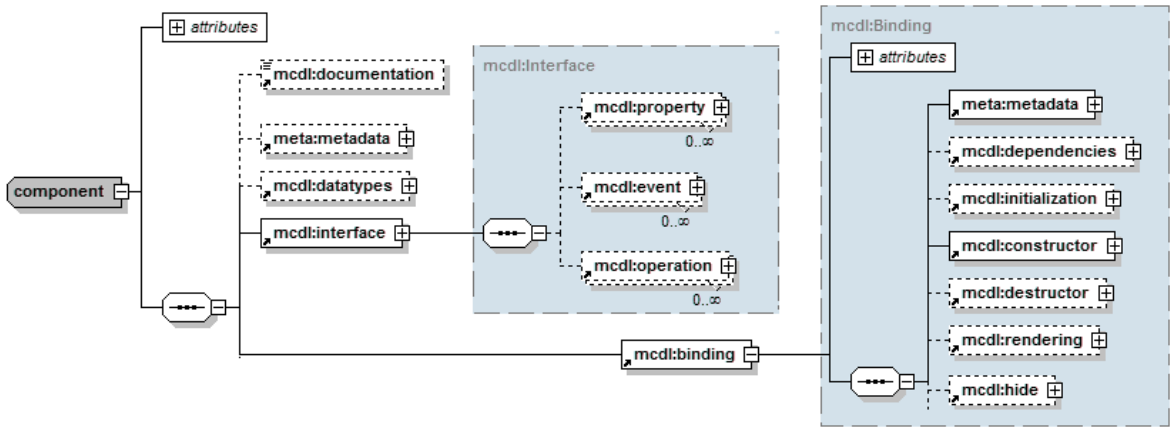

Abb. 5.4: Auszug aus dem XML-Schema einer MCDL-Beschreibung

Abbildung 5.4 zeigt den Aufbau einer MCDL-Beschreibung schematisch mit ihren wichtigsten Elementen. Der *interface*-Teil beschreibt die abstrakte Schnittstelle einer Komponente entsprechend dem vorgestellten Modell, d. h. er enthält die Definition von Properties, Operations und Events sowie der zugehörigen Datentypen. Letztere können direkt im Dokument beschrieben sein oder auf externe Datenschemata verweisen. In dieser einfachsten Form der MCDL wird eine rein syntaktische Typisierung wie in der UISDL unterstellt. Sie erfolgt, in Anlehnung an die bewährten Konzepte der WSDL, in Form von XML-Schema [@Fallside und Walmsley, 2004], was die Kapselung von Web Services als Mashup-Komponenten besonders einfach macht. Die Nutzung von XML erlaubt die Repräsentation von Datentypen in einer einfach serialisierbaren Form, die unabhängig von Programmiersprachen und spezifischen Implementierungen ist. Somit können Komponenten zur Laufzeit unabhängig von

[2] Die UISDL wurde mit anderen Partnern im umgebenden Forschungsprojekt entwickelt und beschränkt sich auf die Beschreibung UI-Komponenten.

ihrer konkreten Umsetzung Daten austauschen. Komponentenentwickler müssen jedoch sicherstellen, dass ihre internen Datenstrukturen so umgewandelt werden, dass sie der externen Schnittstelle entsprechen.

Um später die reichhaltigere Modellierung von Koordination und Synchronisation auf der Präsentationsebene zu ermöglichen, können Ereignissen und Operationen einer Komponente explizit in Beziehung gesetzt werden. Dazu kann in der MCDL jedes Event auf eine *Callback-Operation* und jede Operation auf ein *Callback-Event* verweisen. Abbildung 5.5 veranschaulicht die Funktionsweise: Beim Aufruf einer Operation O_B durch ein Event E_A (mit dem typgleichen Parameter $P_{<X>}$) kann diese Operation einen Rückgabewert bereitstellen, z. B. eine Erfolgs- oder Fehlermeldung. Dieser Rückgabewert $P_{<Y>}$ wird auf das ereignisorientierte Modell abgebildet, also wiederum per Event publiziert – dem Callback-Event E_B. Die Verarbeitung des Rückgabewertes erfolgt durch eine Operation O_A, die durch das aufrufende Event E_A als Callback-Operation referenziert wird. Wenn bei der Modellierung eine bidirektionale Kommunikation vorgesehen wird – d. h. E_A und O_B werden entsprechend verknüpft – so ergibt sich der Rückkanal durch die Referenzen implizit und muss nicht durch den Kompositeur modelliert werden. Diesem ist es aufgrund des *Black-Box*-Charakters ohne weiteres auch nicht möglich, Zusammenhänge zwischen Ereignissen und Operationen einer Komponente zu erkennen.

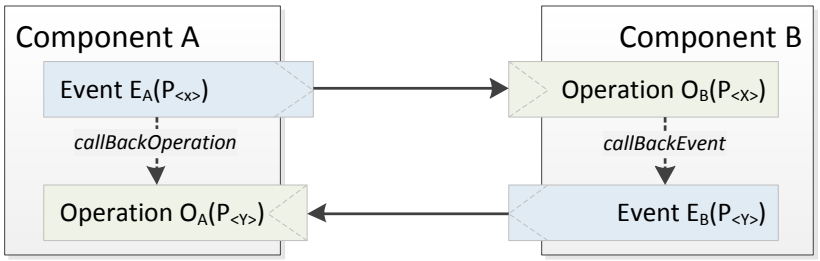

Abb. 5.5: Erweiterte Kommunikation über Callback-Ereignisse und -Operationen

Der Binding-Teil einer MCDL-Beschreibung enthält alle nötigen Informationen, um spezifische Komponentenimplementierungen zu integrieren, instanziieren, initialisieren und über die definierte Schnittstelle anzusprechen. Das abstrakte Komponentenmodell wird dazu an konkrete Methoden der Komponente gebunden, die u. a. die Steuerung des Lebenszyklus' (`constructor`, `destructor`), den Zugriff auf Properties, den Aufruf von Operationen und UI-spezifische Funktionen wie das Ein- und Ausblenden (`rendering`, `hide`) ermöglichen. Dazu wird aus dem Binding auf die Artefakte der Schnittstellendefinition verwiesen, welche unabhängig von der Existenz und Anzahl der Bindings bleiben. Neben der Definition von Metadaten, die zur Auswahl von Bindings benötigt werden, da sie Informationen zur Programmiersprache und -plattform enthalten, ist die Angabe des Konstruktors der einzige obligatorische Teil eines Bindings. Alle restlichen Methoden können gemäß dem Leitprinzip „Convention over Configuration“ gebunden werden. Wird durch den Komponentenentwickler sichergestellt, dass vordefinierte Getter- und Setter-Methoden für Properties vorhanden sind (getProperty(name) und setProperty(name,value)), und sich interne Methoden und Operationen der Schnittstelle in ihrer Benennung und Signatur gleichen, so kann das manuelle Binding entfallen.

Im Idealfall wird davon ausgegangen, dass *Bindings* einer Komponente die spezifizierte Schnittstelle genau umsetzen. Dies ist die notwendige Voraussetzung für deren (kontextadaptiven) Austausch. Im praktischen Einsatz erfolgt jedoch häufig die Integration bestehender Web-Ressourcen und Komponenten, die u. U. eigene Datenrepräsentationen und -modelle unterstellen. Um die Kapselung derartiger Quellen ohne die Implementierung zusätzlicher Adapter zu ermöglichen, können im Binding Transformationsvorschriften hinterlegt werden, die die Abbildung zwischen Schnittstellen und internen APIs erlauben. Das Konzept sieht dafür XSLT-Stylesheets vor, die integriert oder referenziert werden. Alternativ kann die Transformation programmatisch durch Code-Fragmente beschrieben werden, z. B. in JavaScript. Die Interpretation obliegt in beiden Fällen der Laufzeitumgebung.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <component xmlns="http://inf.tu-dresden.de/cruise/mcdl" name="RoutingMap"
3      id="http://inf.tu-dresden.de/cruise/ui/maps/ExtRoutingMap" version="1.1"
4      xmlns:meta="..." xmlns:xs="..."  type="ui">
5      <documentation>...</documentation>
6      <meta:metadata/>
7      <datatypes>
8          <xs:schema targetNamespace="http://vvo-online.de">
9              <xs:complexType name="Location">
10                 <xs:sequence>
11                     <xs:element name="name" type="xs:string"/>
12                     <xs:element name="coordinates" type="vvo:Coordinates"/>
13                 </xs:sequence>
14             </xs:complexType>
15             <!-- additional types ... -->
16         </xs:schema>
17     </datatypes>
18     <interface xmlns:vvo="http://vvo-online.de">
19         <property name="mapType" required="true" type="vvo:MapType"/>
20         <property name="language" type="xml:lang">
21             <default>en-US</default>
22         </property>
23         <!-- additional properties ...  -->
24         <event name="startLocChanged">
25             <parameter name="startLocation" type="vvo:Location"/>
26         </event>
27         <!-- additional events ... -->
28         <operation name="setStartLocation">
29             <parameter name="location" type="vvo:Location"/>
30         </operation>
31         <!-- additional operations ... -->
32     </interface>
33     <binding>
34         <meta:metadata/>
35         <dependencies>
36             <dependency uri="http://core.cruise.org/.../map.js" type="text/javascript"
                   />
37             <!--additional dependencies ... -->
38         </dependencies>
39         <constructor>
40             <code>new RoutingMap();</code>
41         </constructor>
42         <!-- additional binding information ... -->
43     <binding>
44 </component>
```

Lst. 5.1: Beispiel einer Komponentenbeschreibung in MCDL

Codebeispiel 5.1 zeigt eine beispielhafte MCDL-Instanz, die aus Gründen der Übersichtlichkeit nur die wichtigsten Elemente enthält. Zunächst erfolgt im Wurzelelement die Angabe von Namen und eindeutigem Bezeichner der Komponente (Zeile 2/3)

sowie der benötigten Namensräume. Das Attribut `type` zeigt an, ob die Komponente zur Benutzerschnittstelle beiträgt – in diesem Fall setzt die Laufzeitumgebung voraus, dass bestimmte Eigenschaften (z. B. *width*, *height*) und Methoden (*show*, *hide*) von der Komponente implementiert werden. Der Metadatenbereich (Zeile 6) dient der Angabe nicht-funktionaler Informationen, die für die gesamte Komponente gelten, so z. B. Schlüsselworte. Danach folgt die Angabe der verwendeten Datentypen. Im Beispiel wird der Typ Location definiert (Zeile 9–14), der somit zur Typisierung von Eigenschaften und Parametern der Events (Zeile 25) und Operationen (Zeile 29) zur Verfügung steht. Die Schnittstellenbeschreibung (Zeile 18–32) beginnt mit der Deklaration der zustandsgebenden Properties (Zeile 19/20), die als obligatorisch (`required`) gekennzeichnet und mit Standardbelegungen (Zeile 21) versehen werden können. Danach werden die Ereignisse und Operationen der Komponente definiert, die beliebig viele typisierte Parameter enthalten. Potentiell können mehrere Umsetzungen für die Schnittstelle existieren. Das Beispiel beschränkt sich jedoch auf den Regelfall, d. h. die Angabe eines Bindings (Zeile 33–43). Dieses kann zunächst durch Metadaten näher beschrieben werden, z. B. hinsichtlich der Software- und Plattformanforderungen, Lizenz- und Autoreninformationen, usw. Es folgt die Angabe von Abhängigkeiten (Zeile 36), die zur Ausführung der Komponente geladen werden müssen und für webbasierte Anwendungen i. d. R. JavaScript- und Cascading Style Sheets (CSS)-Dateien darstellen. Schließlich wird in Zeile 40 der Konstruktor der Komponente angegeben, an den sich weitere, optionale Binding-Informationen anschließen können.

Ein Nachteil der syntaktischen Typisierung der Schnittstellen ist deren Inflexibilität. Die MCDL unterstellt für die kontextsensitive Suche und den Austausch von Komponenten deren völlige syntaktische Kompatibilität, die im praktischen Einsatz selten gegeben ist. Ein Weg der Flexibilisierung ist die Unterstützung funktionaler Erweiterungen durch die Aufstellung von Subklassenbeziehungen zwischen abstrakten Interfaces. Praktisch bedeutet dies, dass ein Interface I_S einer Komponente S ein bestehendes Interface I_K der Komponente K erweitert, wenn (1) I_K vollständig in I_S enthalten ist, (2) I_S mindestens ein zusätzliches Schnittstellenartefakt (Property, Event, Operation) besitzt, und (3) funktionale Äquivalenz bezüglich der übernommenen Schnittstelle I_K besteht (kurz: $I_K \subset I_S$). Dadurch wird sichergestellt, dass jede Implementierung von I_K ohne Einschränkung durch I_S ersetzt werden kann – die Anzahl möglicher Alternativen bei der Integration der „passendsten“ Komponente steigt. Innerhalb der MCDL kann die Subklassenbeziehung durch ein optionales Attribut im Element `interface` ausgedrückt werden.

Trotz allem erlaubt auch die Nutzung von Interfaces keinerlei syntaktische Variationen an den „vererbten“ Schnittstellenbestandteilen. Als weitaus mächtigere Alternative wird deshalb die semantische Typisierung empfohlen, die im nächsten Abschnitt genauer vorgestellt wird.

5.1.3.3 Semantic MCDL (SMCDL)

Die rein syntaktische Beschreibung einer Schnittstelle ist aus einer Vielzahl von Gründen nicht ausreichend für die Suche und Integration passender Komponenten. Zum einen schlägt die Suche trotz gleicher Semantik schon bei kleinsten syntaktischen Unterschieden zwischen den Schnittstellenbestandteilen (Operations- und

Ereignisnamen, Parametertypen, etc.) fehl. Zum anderen wird deren Funktionalität nicht beachtet, was zu falschen Ergebnissen führen kann. Um die dynamische, kontextabhängige Suche und Integration von Mashup-Komponenten adäquat unterstützen zu können, wird deshalb nach dem Vorbild der SWS (Abschnitt 3.1.2) die Verknüpfung der MCDL mit semantischen Konzepten ermöglicht. Grundlage bieten die Annotationskonzepte von SAWSDL und WSMO-Lite [Kopecký und Vitvar, 2008], die mit Erweiterungen auf die MCDL angewendet werden. Dadurch können Schnittstellenbestandteile mit semantischen Konzepten verknüpft werden, die z. B. Datentypen, Nutzer- und Kontextmodelle sowie Funktionalitäten repräsentieren.
Abbildung 5.6 zeigt die Annotationsmöglichkeiten von Semantic MCDL (SMCDL) schematisch. Es wird ersichtlich, dass – je nach Schnittstellenbestandteil – die semantische Beschreibung mit verschiedenen Intentionen erfolgen kann.

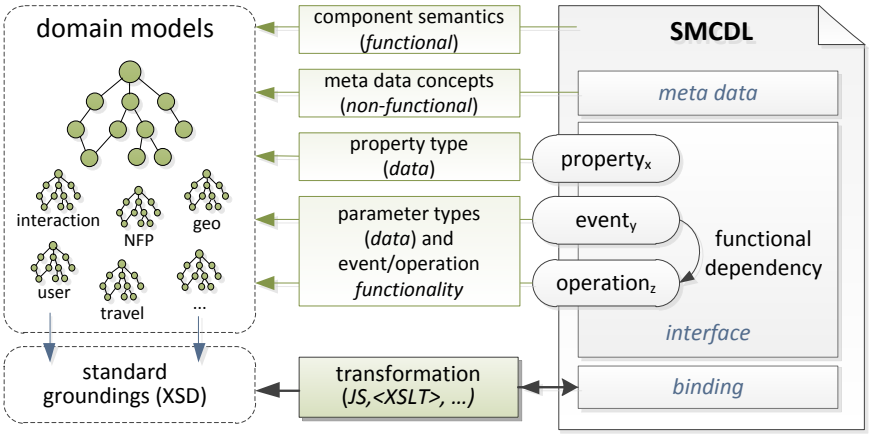

Abb. 5.6: Semantische Annotation der Komponentenbeschreibungen in der SMCDL

Semantische Annotationen können prinzipiell einer von drei Kategorien zugeordnet werden: (1) Die *Datensemantik* dient der Typisierung von Properties und Parametern mit semantischen Konzepten, wobei nur eine Annotation pro Element erlaubt ist. (2) Die *funktionale Semantik* von Komponenten, Operationen und Events kann durch mehrere, als komplementär zu wertende Modellreferenzen, d. h. Verweise auf funktionale Konzepte bzw. *Tasks*, an den entsprechenden Elementen annotiert werden. (3) Zudem können die Metadaten einer Komponente durch Verweise auf Basiskonzepte *nicht-funktionaler Semantik*, wie Autor, Preis und Lizenz, qualifiziert werden.
Redundanzen innerhalb der Annotationen, wie Äquivalenzen oder Vererbungsrelationen zwischen Konzepten, sollten durch Autorenwerkzeuge vermieden werden, da sie einen erhöhten Aufwand bei der Suche und Rangfolgebildung verursachen.

Datensemantik

Wie in Abschnitt 3.1.2 dargelegt, bildet die semantische Typisierung von Ein- und Ausgaben die Basis für dynamische Kompositionsansätze und die Bindung von Diensten bzw. Komponenten zur Laufzeit. Entsprechend können auch alle öffentlichen Schnittstellenkonzepte von Mashup-Komponenten, wie Parameter von Events und Operations sowie Properties, mit Ontologiekonzepten (z. B. Location), Relationen oder Datatype-Properties (z. B. hasID) typisiert werden. Hierzu dient das Attribut `type` der MCDL, wie Listing 5.2 zeigt.

Die Definition des Events (Zeile 3–5) unterscheidet sich nicht von der Deklaration in MCDL (vgl. Abbildung 5.1, Zeile 24–26). Statt auf einen XML-Schema-Datentyp verweist der Typ vvo:Location des einzigen Parameters hier allerdings auf das Konzept Location einer externen Ontologie, die über den Namespace vvo definiert ist.

```
1  <property name="currentLocation" required="false" type="vvo:Location"/>
2  ...
3  <event name="startSelected">
4      <parameter name="startLocation" type="vvo:Location"/>
5  </event>
```

Lst. 5.2: Semantische Typisierung einer Property

Die Kommunikation zwischen Komponenten zur Laufzeit kann vor diesem Hintergrund verschieden ablaufen. Entweder es werden tatsächlich semantische, z. B. RDF-Daten, ausgetauscht, oder für die referenzierten semantischen Konzepte existieren standardisierte *Groundings*, z. B. in XML-Schema, wie in der Abbildung angedeutet. In diesem Fall müssen Grounding-Mechanismen wie bei SAWSDL zum Einsatz kommen. Die Kommunikation auf semantischer Ebene ist insbesondere für Projekte relevant, die auf einer semantischen Datenbasis arbeiten. Im Rahmen dieser Arbeit wird jedoch ein XML-basiertes Grounding genutzt. Neben der Plattform- und Sprachunabhängigkeit spricht insbesondere die weite Verbreitung und Nutzung von XML im Web für dessen Nutzung. Nicht zuletzt erfolgt die Beschreibung der Datenschnittstellen klassischer Web Services in WSDL über XML-Schema, und auch das in REST Services stark verbreitete JSON-Format kann in eine XML-Repräsentation umgewandelt werden.
Das Konzept der semantischen Typisierung der Datenschnittstelle umfasst also sowohl semantische Domänenmodelle als auch XML-Schema-Repräsentationen ihrer Konzepte, die plattformübergreifend durch eine Modellverwaltungsinstanz (vgl. TyRe, Abschnitt 6.2.1) bereitgestellt werden. Zur Umwandlung müssen Transformationsvorschriften in beide Richtungen vorliegen: eine zur Abbildung von XML-Instanzdokumenten auf die semantische Ebene (*Lifting*), d. h. in einen RDF-Graphen, und eine für die umgekehrte Richtung (*Lowering*). Wie diese Abbildung erfolgt, ist konzeptionell nicht vorgeschrieben. Alternativen hierfür, wie die automatische Generierung von XML-Schemata, wurden in Abschnitt 3.1.2 vorgestellt.
Die semantische Typisierung bildet die Grundlage für die *Mediation* von Daten, die sich an den Konzepten von SAWSDL ausrichtet (vgl. Abbildung 3.7) und aus Laufzeitsicht genauer in Abschnitt 6.2.4.2 beleuchtet wird. Hierbei dienen die genutzten Ontologien als Verbindungsglied zwischen den Daten, was im Umkehrschluss die Nutzung einheitlicher, semantischer Modelle impliziert. Allerdings verträgt sich das Konzept auch mit dem Einsatz verschiedener Domänenontologien, sofern Ontology-Mapping-Verfahren zur Anwendung kommen. Derartige Konzepte – Noy (2009) bietet eine ausführliche Betrachtung – ermöglichen den semantischen Modellabgleich, stehen allerdings nicht im Fokus dieser Arbeit.
Bei der Kapselung existierender Komponenten und Dienste kann die Nutzung von Standard-Groundings nicht unterstellt werden. Deshalb können die bereits beschriebenen Transformationsvorschriften genutzt werden, um innerhalb einer Binding-Definition die Abbildung interner Datenmodelle auf die Groundings zu beschreiben. Codebeispiel 5.3 zeigt eine exemplarische Vorschrift und deren Verknüpfung mit einem semantischen Datentyp. Falls im obigen Beispiel der generische Typ

travel:Location genutzt werden soll, so muss der dienstspezifische Typ vvo:Location darin umgewandelt werden. Hier erfolgt dies durch die Referenzierung eines XSLT-Stylesheets (Zeile 4), welches auf alle von der Komponente bereitgestellten Daten angewendet wird, die mit dem Typ travel:Location verknüpft sind (Zeile 5). Auch für eingehende Daten muss eine entsprechende Transformation angegeben werden. Derartige Vorschriften sind optional und entfallen – im Gegensatz zum Konzept der SAWSDL – bei der Nutzung der Standard-Groundings.

```
1 <binding>
2     ...
3     <transformation xsi:type="ToStdGroundingTransformation">
4         <stylesheet href="http://vvo-online.de/transf/location.xsl" type="application/
              xslt+xml"/>
5         <datatype ref="travel:Location"/>
6     </transformation>
7 </binding>
```

Lst. 5.3: Beispiel einer Abbildungsvorschrift auf das Standard-Grounding in SMCDL

Da die semantische Typisierung von Properties und Parametern die syntaktische Typisierung der MCDL ersetzt, ist sie obligatorisch. Falls sich die semantischen Annotationen der SMCDL allein auf die Angabe von Datensemantik beschränken, so entspricht die Mächtigkeit der Beschreibung der MCDL, entsprechend Abbildung 5.3. Zur Verbesserung der Suchergebnisse bei der dynamischen Bindung ist allerdings die zusätzliche Angabe funktionaler und nicht-funktionaler Semantik vielversprechend. Die nächsten Abschnitte geben Aufschluss über die entsprechenden Konzepte.

Funktionale Semantik

Zur Unterstützung der Suche von Mashup-Komponenten nach funktionalen Eigenschaften wird eine Beschreibung der entsprechenden Semantik mittels eines *bottom-up*-Ansatzes vorgeschlagen (vgl. Abschnitt 3.1.2). Im Gegensatz zu den schwergewichtigen und komplexen Ansätzen WSMO und OWL-S wird hier der Leichtgewichtigkeit von Mashup-Anwendungen Rechnung getragen. Die semantische Annotation lässt sich direkt in die Beschreibungssprache MCDL integrieren und unterstützt somit das zustandsbasierte Komponentenmodell.

Die Annotation von Operationen und Ereignissen mit funktionaler Semantik erfolgt – in Anlehnung an WSMO-Lite [Kopecký und Vitvar, 2008] – über Modellreferenzen im Attribut `functionality`. Es enthält Verweise auf eines oder mehrere domänenspezifische Konzepte, welche funktionale Kategorien oder – angelehnt an den Ansatz von CoSMoS [Fujii und Suda, 2009] – Aktionen repräsentieren. Dieses Vorgehen ist nicht nur intuitiver und praktikabler als die rein zustandsbasierte Modellierung, sondern ermöglicht auch die automatische Suche von Komponenten auf Basis von Tasks bzw. Task-Modellen, wie von Tietz et al. (2011) nachgewiesen wurde. Eine weitere Möglichkeit der Verwendung der Annotationen zeigt das Forschungsprojekt Mefisto [@Mefisto, 2011] auf. Für einen gewählten Datensatz werden Nutzern der Anwendung *VizBoard* adäquate Visualisierungskomponenten empfohlen, wofür sowohl die Daten- als auch die funktionale Schnittstellenbeschreibung auf Basis von SMCDL genutzt wird. Das o. g. Attribut dient in diesem Fall dem Verweis auf Konzeptualisierungen einer Visualisierungsontologie, wie näher durch Voigt et al. (2012a) erläutert wird.

Codebeispiel 5.4 zeigt am Ausschnitt einer SMCDL-Beschreibung, wie die funktionale Annotation von Ereignissen (Zeile 1) und Operationen (Zeile 6) erfolgt.

```
1  <event name="routes" functionality="tsk:RoutesCalculated"
2                       trigger="operation" dependsOn="tsk:RouteSelection">
3      <parameter name="result" type="travel:Routes"/>
4  </event>
5  ...
6  <operation name="setStart" functionality="tsk:RouteStartSelection">
7      <parameter name="location" type="travel:Location"/>
8  </operation>
9  <operation name="setDestination" functionality="tsk:RouteDestinationSelection">
10     <parameter name="location" type="travel:Location"/>
11 </operation>
12 ...
```

Lst. 5.4: Funktionale Annotation von Ereignissen und Operationen

Gegenüber Web Services besitzen UI-Komponenten mit der Benutzerschnittstelle einen zusätzlichen Ein-/Ausgabekanal. Somit können sie auch Funktionalitäten bieten, die nicht durch Events angezeigt oder durch Operationsaufrufe ausgelöst werden, z. B. die Sortierung und Filterung von Einträgen einer Liste. Zur Modellierung derartiger Funktionalitäten können auch dem Wurzelelement `component` über das Attribut `functionality` mehrere Modellreferenzen hinzugefügt werden.

Die Nutzung von Ontologien in Beschreibungslogik erlaubt es, komplexe funktionale Zusammenhänge zu erkennen und auf ihrer Basis kompatible Komponenten zu finden. In Codebeispiel 5.4 kann geschlussfolgert werden, dass die Operationen setStart (Zeile 6) und setDestination (Zeile 9) gemeinsam die Grundlage zur Routenberechnung bilden. Eine entsprechende Anfrage nach dem funktionalen Konzept RouteSelection könnte durch die skizzierte Komponente erfüllt werden, vorausgesetzt, dass die Parameter übereinstimmen und RouteSelection als Schnittmenge der Konzepte RouteStartSelection und RouteDestinationSelection modelliert ist.
Oftmals bestehen zwischen Operationen und Ereignissen einer Komponente implizite Zusammenhänge, z. B. wenn eine Operation zu einer internen Zustandsänderung führt, welche wiederum durch ein Event nach außen publiziert wird. Für den Such- und Auswahlprozess von Komponenten können derartige Zusammenhänge dann von Relevanz sein, wenn bestimmte Events benötigt, aufgrund nicht verdrahteter Operationen aber niemals ausgelöst werden. Deshalb können durch das optionale Attribut `trigger` die Auslösemöglichkeiten eines Events angegeben werden: *interaction* impliziert Nutzerinteraktionen, *operation* die Publikation in Folge von Operationsaufrufen, und *internal* in Folge interner Verarbeitungsroutinen, z. B. in Folge eines Serviceaufrufs. Im Fall von *operation* verweist das Attribut `dependsOn` auf eine Menge funktionaler Kategorien, die durch entsprechende Operationen erfüllt sein müssen. In obigem Beispiel zeigt die Annotation in Zeile 1 die Abhängigkeit von Operationen, die die Routenangabe (RouteSelection) erlauben – das Event hängt also implizit vom Aufruf der beiden angegebenen Operationen ab. Diese Vorgehensweise hat gegenüber der direkten Referenzierung von Operationen den entscheidenden Vorteil, dass sie feingranularer ist und somit auch funktioniert, wenn eine Operation mehrere Funktionalitäten anbietet.
Die vorgestellten Annotationsformen sind vielseitig nutzbar, u. a. zur qualitativen Verbesserung der Treffermenge bei der dynamischen Suche und Integration von

Komponenten, aber auch zur Empfehlung von Komponenten auf Basis von Task-Modellen und Visualisierungswissen. Hierfür können beliebige Domänenontologien die Grundlage bilden, deren Erstellung jedoch nicht im Fokus der Arbeit steht.

Nicht-funktionale Semantik

Während funktionale Eigenschaften und Zusammenhänge die Suche nach Komponenten an sich ermöglichen, bedarf es weiterer Informationen, um passende Kandidaten zu priorisieren und eine sinnvolle, kontextabhängige Auswahl zu unterstützen. Das zugrunde liegende Vokabular bieten Ontologien, deren Klassen, Relationen und Individuen auf den Metadaten-Teil der SMCDL abgebildet werden. Die verfügbaren Elemente und Attribute werden bezüglich ihrer Ausprägungen und Wertebereiche durch das semantische Modell eingeschränkt. Zusätzlich kann der Verweis auf spezifische semantische Konzepte erfolgen. Codebeispiel 5.5 zeigt dies beispielhaft für den Preis einer Komponente (Zeile 5). Statt einer beliebigen Zeichenkette erfolgt für die Währung die Referenzierung eines Individuums der Ontologieklasse Currency durch eine entsprechende URI. Dieses Vorgehen bietet eine Reihe von Vorteilen – neben der Eindeutigkeit referenzierter Konzepte in erster Linie die Erweiterbarkeit der Instanzmenge. So können jederzeit neue Währungen hinzugefügt werden, ohne dass Anpassungen am SMCDL-Schema nötig werden.

```
1 <meta:metadata>
2     <meta:keywords>map, route</meta:keywords>
3     <meta:license id="nfp:LGPLv3" />
4     <meta:price>
5         <meta:absolutePrice currency="nfp:EUR" value="10" />
6     </meta:price>
7     <meta:nfp type="qos:Availability" value="99" unit="m:percent"/>
8 </meta:metadata>
```

Lst. 5.5: Angabe von Metadaten in einer SMCDL-Instanz

Um die Unabhängigkeit zwischen den genutzten Modellen und der SMCDL auch für Metadaten zu wahren, können über das generische Element `nfp` beliebige Konzepte weiterer Modelle unter Angabe von Typ, Wert, und einer optionalen Einheit referenziert werden (Zeile 7). Die Ausprägung der genutzten Ontologie(n) ist konzeptionell nicht vorgeschrieben, und ihre Entwicklung steht nicht im Fokus der Arbeit. Neben den vordefinierten Konzepten können bewährte Modelle aus dem Bereich SWS nachgenutzt werden, z. B. das Modell von O'Sullivan (2006), welches auch in WSMO [Fensel et al., 2008] Anwendung findet. Es bietet ein umfangreiches Vokabular zur Definition nicht-funktionaler Aspekte für Web Services, die zumindest in Teilen auch für Mashup-Komponenten Relevanz besitzen.
Da Mashup-Komponenten der Präsentationsebene im Gegensatz zu konventionellen Diensten nicht entfernt ausgeführt, sondern auf dem Client dynamisch integriert werden, ist die Beschreibung ihrer Soft- und Hardware-Anforderungen nötig. Auch die Annotationen mit Interaktionstechniken und Modalitäten kann auf diese Art und Weise vorgenommen werden. In beiden Fällen muss auf semantische Modelle verwiesen werden, die Individuen zur Repräsentation von Ausführungsplattformen und Endgeräteklassen sowie Modalitäten, Gesten, Systemfeedback usw. bereitstellen. Die semantischen Annotationen nicht-funktionaler Art unterstützen zur Laufzeit das *Ranking*, d. h. die kontextspezifische Sortierung und Auswahl funktional passender Kandidaten. Der damit verbundene Auswahlprozess wird in Abschnitt 6.1 vorgestellt.

5.1.3.4 Mashup Component Description Ontology (MCDO)

Durch Yu et al. (2007) wurden diverse, allgemeingültige Anforderungen an Komponentenbeschreibungen formuliert, u. a. dass sie formal, einfach, menschenlesbar und modular sein sollen. All diese Eigenschaften werden in besonderem Maße durch Ontologien abgedeckt [Paulheim, 2009]. Vor dem Hintergrund der angestrebten semantischen Suche von Komponenten ist es somit naheliegend, sie direkt als Instanzen einer Ontologie – der Mashup Component Description Ontology (MCDO) – zu beschreiben. Die SMCDL bildet somit den Kompromiss aus der Komplexität der semantischen und der Einfachheit der syntaktischen Beschreibung.

Die MCDO stellt alle bislang besprochenen Konzepte der SMCDL zur Verfügung. Darüber hinaus bietet sie eine Reihe zusätzlicher Beschreibungsmittel. Codebeispiel A.2 im Anhang zeigt einen Ausschnitt der Serialisierung einer Beispielinstanz in RDF/XML.

Zum einen besteht die Möglichkeit, *Vorbedingungen* an die Funktionalität einer Operation zu stellen. Analog zum Verständnis in WSMO [Fensel et al., 2008] dienen sie dazu, den Kontext genauer zu spezifizieren, in dem eine Funktionalität garantiert erbracht werden kann. Falls sich die Möglichkeiten zur Routenberechnung durch die vorgestellte Komponente beispielsweise auf Deutschland beschränken, so kann dies durch eine Vorbedingung definiert werden. Somit wird die Präzision der Suche und Auswahl passender Komponenten in einem bestimmten Kontext verbessert.

Zur Modellierung besitzt die MCDO in Anlehnung an OWL-S die generische Klasse Precondition. Diese kann mit einer oder mehreren Modellreferenzen assoziiert sein. Konkrete Vorbedingungen können durch Subklassen gestellt werden, z. B. als SPARQLPrecondition, die aufgrund ihrer plattformunabhängigen und standardisierten Syntax von SPARQL im späteren Prototyp Verwendung findet.

Die ontologiebasierte Repräsentation erlaubt zudem die dynamische Berechnung nicht-funktionaler Eigenschaften unter Einbeziehung von Kontextinformationen. Mit Hilfe von Regeln wird dabei entweder ein statischer Wert überschrieben, oder gänzlich neue Metadaten generiert. Der in Codebeispiel 5.5 definierte Preis stellt einen typischen Anwendungsfall dar – dieser kann in Abhängigkeit vom Nutzerkontext (*Premiumaccount*) variieren.

In der MCDO können beliebig viele derartiger Regeln über die Eigenschaft mit der Klasse MetaData bzw. komponentenspezifischer Instanzen davon verbunden werden. Konkrete Regelsprachen sind wie im Fall der Vorbedingungen durch Subklassen repräsentiert – im vorliegenden Prototypen werden beispielsweise *JenaRules* [@Jena, 2011] verwendet. Codebeispiel 5.6 zeigt eine solche Regel, die – angewendet auf die zugehörige Komponente – deren Preis auf 10 setzt, falls eine nicht näher beschriebene SPARQL-Anfrage als wahr ausgewertet wird. In Zeile 1 ist u. a. zu sehen, wie die Verknüpfung mit Kontextinformationen über das *Built-In* queryContext erfolgt. Dabei handelt es sich um eine Erweiterung, die in Java umgesetzt wurde und die Anfragen an das Kontextmodell des Nutzers ermöglicht.

```
1 [inferPriceRule1:  queryContext("SPARQL-ASK"), ?price a nfp:AbsolutePrice ->
2     ?price nfp:hasValue 10]
```

Lst. 5.6: Jena Rule zur dynamischen Berechnung des Preises einer Komponente

Die drei aufeinander aufbauenden Beschreibungssprachen geben Komponentenentwicklern die Möglichkeit, Mashup-Komponenten in einfacher und verständlicher Form nach dem vorgestellten Modell universell zu beschreiben. Neben der Deklaration der rein syntaktischen Schnittstellen über MCDL erlaubt die Beschreibung mit SMCDL die Verknüpfung von Schnittstellenbestandteilen mit semantischen Domänenkonzepten. Dies ermöglicht zum einen die Abstraktion von syntaktischen Datentypen und zum anderen die Qualifikation mit funktionaler und nicht-funktionaler Semantik zur Unterstützung des späteren Such- und Auswahlprozesses. Die ontologiebasierte Repräsentation auf Basis der MCDO bietet schließlich erweiterte Möglichkeiten, wie die Definition von Vor- und Nachbedingungen sowie von kontextadaptiven Metadaten. Eine Mashup-Umgebung ist jedoch immer nur so gut, wie die Komponenten, die es unterstützt [Gartner, 2007]. Deshalb wird im nächsten Abschnitt erläutert, wie deren Entwicklung oder Kapselung nach dem beschriebenen Modell erfolgen sollte.

5.1.4 Nutzung der Konzepte zur Komponentenentwicklung

Die Verfügbarkeit möglichst vieler durchdachter und robuster Komponenten spielt für die Mashup-Komposition eine ähnlich tragende Rolle, wie die vergleichbarer Web Services für darauf basierende Geschäftsprozesse. Wie bereits angeklungen ist, lässt sich die Anwendungskomplexität durch die angestrebten Abstraktionen nicht auflösen. Zwar wird die Modellierung und Komposition von Anwendungen stark vereinfacht – dies bedeutet aber, dass die Entwicklung von Komponenten besonders sorgfältig erfolgen muss. Der folgende Abschnitt widmet sich diesem Aspekt.
Bei der Spezifikation der Komponentenschnittstelle ist auf deren Wiederverwendbarkeit zu achten. Dazu zählt die Nutzung möglichst generischer Operationen, Ereignisse und Datentypen. Die Implementierung sollte hingegen robust und fehlertolerant gestaltet sein. Nicht nur in dieser Hinsicht unterliegen Mashup-Komponenten den gleichen Gestaltungs- und Entwicklungsregeln wie Web Services (vgl. [Papazoglou und Yang, 2002]) und Komponenten des CBSE.
Für die Erstellung von Komponenten wird eine API bereitgestellt, die unabhängig von Programmiersprachen oder Plattformen ist. Aus Sicht der Laufzeitumgebung dient sie dem einheitlichen Zugriff auf Komponenten, z. B. durch Methoden zur Initialisierung, zum Setzen und Auslesen von Properties, für den Aufruf von Operationen usw. Anders herum erlangen Komponentenentwickler dadurch Zugriff auf Objekte der Umgebung, die z. B. ihre Events entgegennehmen oder den Aufruf entfernter Dienste erlauben. Es obliegt folglich dem Entwickler, Komponenten entsprechend der API zu erstellen. Die Mehrzahl der Abbildungen zwischen Schnittstelle und Implementierung erfolgt über Konventionen. So werden die Initialisierung einer Komponente über deren Methode init und der Zugriff auf Properties über get/setProperty realisiert. Für Abweichungen von den Konventionen stehen Entwicklern die vorgestellten Konzepte der MCDL zur Abbildung interner APIs auf das Komponentenmodell zur Verfügung. Hier können beispielsweise dedizierte Zugriffsmethoden spezifiziert oder Datentyptransformationen referenziert werden.

Neben der Implementierung neuer Komponenten stellt die Nutzung bestehender Ressourcen und Anwendungen den häufigsten Anwendungsfall im Kontext von Mashups dar. Zu diesem Zweck können und sollten Autorenwerkzeuge und Mechanismen zur „Komponentisierung“ zum Einsatz kommen.

Komponentenmodell und -beschreibung sind so gehalten, dass die Kapselung von Web Services, die mittels WSDL oder der Web Application Description Language (WADL) beschrieben sind, ohne weiteres möglich ist. Die Abbildung ihrer Schnittstellenbestandteile auf MCDL kann automatisiert erfolgen. Jeder Web-Service-Operation wird eine Operation der Komponente zugeordnet, und jede Rückgabe einer Operation entspricht einem Event. Beide können über das `dependsOn`-Attribut in Beziehung gesetzt werden. Ebenso ist die Abbildung semantischer Dienstbeschreibungen wie SAWSDL auf SMCDL möglich, wobei die semantischen Annotationen direkt übertragen werden. Im Rahmen dieser Arbeit wurde die Machbarkeit der Kapselung exemplarisch für WSDL-basierte Dienste gezeigt. Die Komponenten für den Dienstzugriff werden dabei dynamisch durch die Laufzeitumgebung aus deren Schnittstellenbeschreibung generiert. Der Ansatz ist in seiner Komplexität allerdings auf funktional unabhängige Operationen beschränkt. Sobald komplexere Authentifizierungsmechanismen, wie das Abfragen und Mitführen von API-Keys, nötig werden, muss die Dienstkapselung manuell erfolgen.
Im Hinblick auf die Nutzung weiterer Web-Ressourcen wurde die automatische Kapselung bestehender Web-Komponenten mit Fokus auf den UI-Bereich (Widget Toolkits und Frameworks) konzipiert und getestet [Kreska, 2008]. Das Konzept beinhaltet konfigurierbare Wrapper, die Ressourcen verschiedener Technologien (u. a. MXML/Flash, JavaScript und Java Applets) zur Entwicklungszeit als Komponenten kapseln und die entsprechenden Beschreibungen generieren.

Das vorgestellte Komponentenmodell erlaubt die Kapselung und Beschreibung wiederverwendbarer Bestandteile einer kompositen Webanwendungen auf eine universelle und technologieunabhängige Art. Die eigentliche Entwicklung der kompositen Anwendungen besteht in der bedarfsgerechten Kombination dieser Komponenten. Dazu müssen die funktionalen, zeitlichen und örtlichen Zusammenhänge zwischen ihnen definiert werden. Das folgende Unterkapitel widmet sich der Vorstellung des entsprechenden plattformunabhängigen Vokabulars in Form eines belangorientierten Metamodells.

5.2 Ein belangorientiertes Metamodell für interaktive Mashup-Anwendungen

Um Anwendungen modellgetrieben als Komposition universeller „Bausteine“ entwickeln zu können, bedarf es eines plattformunabhängigen Modells, welches alle Belange einer lauffähigen Anwendung beschreibt. Letztere umfassen u. a. die genutzten Komponenten, die Kommunikationsbeziehungen zwischen ihnen, die Anordnung von UI-Komponenten auf der Oberfläche, Informationen zur Navigation sowie zum adaptiven Verhalten der Komposition zur Laufzeit. Zu diesem Zweck wurde ein erweiterbares Metamodell entwickelt – das *Mashup Composition Model* – welches all diese Aspekte beschreibt.
Wie Abbildung 5.7 zeigt, folgt der hier vorgestellte Ansatz der Meta Object Facility (MOF) Spezifikation [@OMG, 2006]. Das im Folgenden präsentierte Metamodell ist darin auf MOF Ebene M2 verortet, während seine Instanzen auf Ebene M1 konkrete Anwendungsmodelle darstellen. Die Konformität zu MOF stellt sicher, dass alle MOF-konformen Systeme die Modelle interpretieren und weiter verarbeiten können.

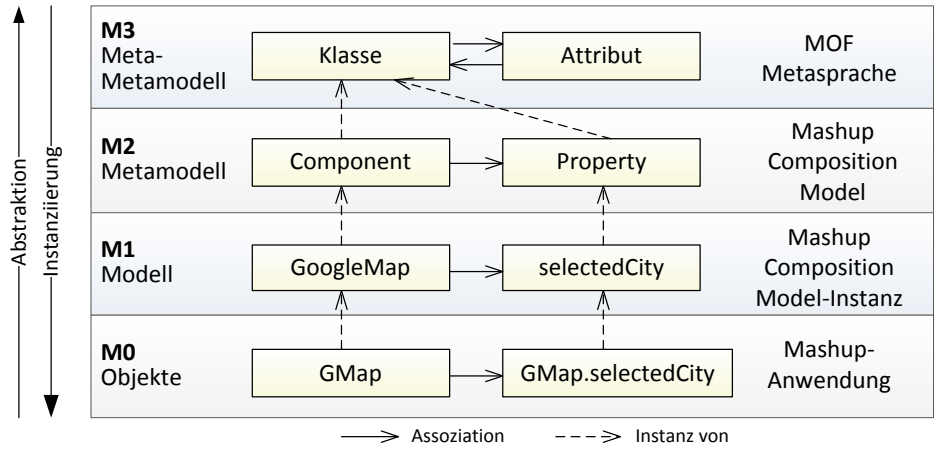

Abb. 5.7: Einordnung des Kompositionsmetamodells in die MOF-Architektur

Abbildung 5.8 gibt einen Überblick über die obligatorischen Teilmodelle des Kompositionsmodells: das *Conceptual*, das *Communication*, das *Layout* und das *Screen Flow* Model. Jedes der Teilmodelle beschreibt einen spezifischen Anwendungsaspekt, vergleichbar mit den Teilmodellen aus WebML und UWE.

Die Abhängigkeiten zwischen diesen Belangen sind aus der Abbildung ersichtlich: Zur Verknüpfung von Komponenten im Communication Model wird beispielsweise auf die entsprechenden Vertreter im Conceptual Model verwiesen. Im Gegensatz zu konventionellen Web-Engineering-Ansätzen erfolgt zudem nicht die Modellierung von Daten, sondern von funktionalen Anwendungsbausteinen. Diese können, wie im vorigen Abschnitt beschrieben, semantisch annotiert werden, wodurch der Bezug zu externen Domänenmodellen bzw. deren Konzepten hergestellt wird.

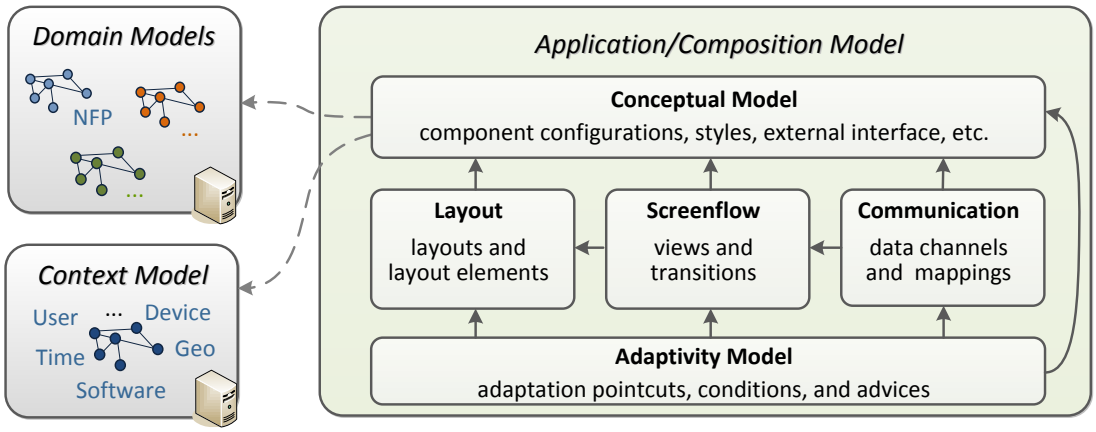

Abb. 5.8: Überblick sowie Domänen- und Kontextbezug des Kompositionsmodells

Abbildung 5.9 veranschaulicht die Zusammenhänge aus der Sicht der Modellierung. Dabei wird ersichtlich, wie anhand des *Adaptivity Model*s exemplarisch die Erweiterbarkeit um Teilmodelle zur Beschreibung zusätzlicher Anwendungsbelange – in diesem Fall adaptives Verhalten – erfolgt. Grundlage dafür bietet die abstrakte Modellklasse MashupModel, deren Subklassen neue Aspekte, wie Qualitätsanforderungen und Kollaborationsmöglichkeiten, im Modell repräsentieren können.

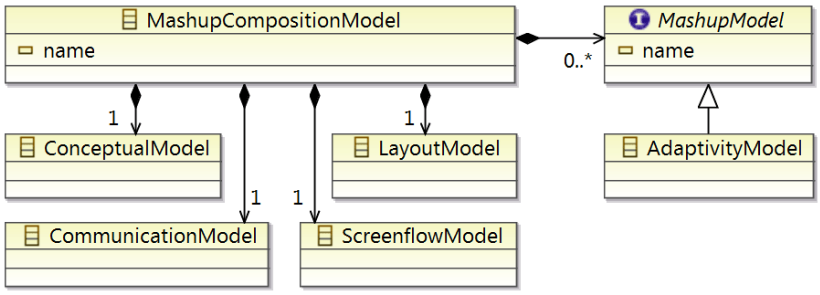

Abb. 5.9: Das *Mashup Composition Model* und seine Teilmodelle

Die folgenden Abschnitte beschreiben die Teilmodelle und ihre Bezüge im Detail, wobei sich die Abbildungen auf die wichtigsten Klassen und Attribute beschränken und einige Vereinfachungen enthalten, um die Übersichtlichkeit zu wahren.

5.2.1 *Conceptual Model* – Modellierung der Anwendungskonzepte

Das Conceptual Model beinhaltet alle anwendungsweiten Konzepte, die i. d. R. aus anderen Teilmodellen referenziert werden. Auf oberster Ebene besteht es aus vier Containern für visuelle Stile (Styles), Komponenten (Components), Datentypen (DataTypes) sowie Informationen zur äußeren Schnittstelle zur Umgebung (Environment), wie Abbildung 5.10 mit einigen Vereinfachungen verdeutlicht.

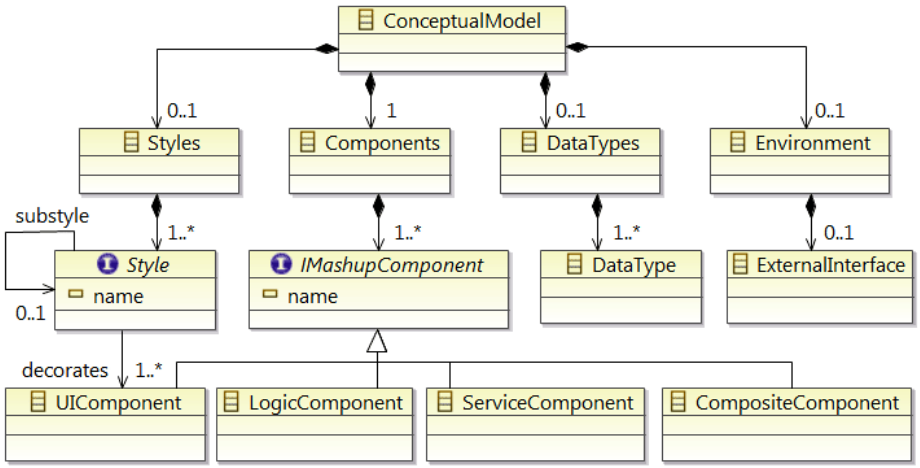

Abb. 5.10: Modellierung der Basiskonzepte im *Conceptual Model*

Die wichtigsten Bestandteile des Modells stellen die zu integrierenden Komponenten dar, die in konkreter Form, d. h. mit Verweis auf eine Implementierung, oder abstrakt in Form eines „Templates“ repräsentiert werden können. Der nächste Abschnitt stellt die Modellierungsmöglichkeiten genauer vor.

5.2.1.1 Komponenten und Templates

Die Klasse Components stellt einen zentralen Bestandteil des Modells dar und enthält alle funktionalen Bausteine bzw. **Komponenten** einer Anwendung. Diese entsprechen dem vorgestellten Komponentenmodell (vgl. Abschnitt 5.1), welches vollständig über die Klasse IMashupComponent im Kompositionsmodell repräsentiert

ist. Die Referenzierung von Komponenten erfolgt über die *ID* und *Version* aus ihrer Beschreibung. Der Typisierung von Schnittstellenbestandteilen dienen die Klassen Datatype zur Einbettung syntaktischer Schemata und ModelReference (nicht in der Abbildung) für den Verweis auf externe, semantische Modelle.

Auch wenn alle Komponenten den gleichen Prinzipien entsprechen, so wird im Kompositionsmodell unterschieden, ob sie auf der Präsentationsebene angesiedelt sind oder nicht. Diese Information steht in der MCDL durch das Attribut `type` bereit (vgl. Abschnitt 5.1.3.2) und zieht die entsprechende Modellierung nach sich: UI-Komponenten werden als Instanz der Klasse UIComponent, alle weiteren Komponenten als ServiceComponent repräsentiert. Die Spezialisierung ermöglicht die gesonderte Behandlung von Komponenten der Präsentationsebene im Modell, z. B. bei der Definition von Layout und Sichten, und auch zur Laufzeit, da an sie erweiterte Anforderungen, wie Methoden zum Ein- und Ausblenden, gestellt werden.

Um die Interoperabilität zwischen Komponenten bzw. ihren Events und Operations zu erhöhen, ermöglichen Logik-Komponenten (LogicComponent) Datenmanipulationen wie Filterung, Konvertierung und Aggregation. Diese vordefinierten Klassen repräsentieren somit funktionale Möglichkeiten der Laufzeitumgebungen im Modell. Gerade der Austausch komplexer Daten, wie Person, zwischen Komponenten macht es nötig, Teile davon (Vorname, Nachname, Geschlecht) für andere Komponenten verfügbar zu machen. Logik-Komponenten ermöglichen dazu die Aufteilung und Zusammenführung von Events (EventSplit, EventJoin) oder die Iterationen über Listen von Daten (Iterator). Dadurch können Events und Operations auch in für Komponentenentwickler unvorhersehbaren Kombinationen verknüpft werden. Die Mächtigkeit der modellierbaren Transformationen beschränkt sich – auch aus Gründen der Nachvollziehbarkeit – auf die Parameterebene. Kompatibilität auf Datenebene wird zur Laufzeit durch die semantische Mediation sichergestellt, die in Abschnitt 6.2.4.2 beschrieben ist.

Kurz zusammengefasst binden Service-Komponenten beliebige externe Dienste an, deren Daten durch Logik-Komponenten transformiert und durch UI-Komponenten schließlich visualisiert und manipuliert werden können. Neben diesen atomaren Komponententypen kann das Modell auch komposite Komponenten enthalten, die ihrerseits Kompositionen darstellen.

Templates sind eine abstrakte Repräsentationsform von Komponenten im Kompositionsmodell. Sie repräsentieren keine konkrete Komponentenimplementierung, sondern dienen der Laufzeitumgebung als Zielbeschreibung für die Discovery in Sinne der SWS (vgl. Abschnitt 3.1.2). Templates stellen somit „ideale“ Komponenten einer Komposition dar, die in der Praxis freilich häufig in Anlehnung an oder auf Basis von bestehenden Komponenten modelliert werden. Ihr Abstraktion erfolgt mit dem Ziel der Plattformunabhängigkeit und Kontextadaptivität der Anwendung.

Wie Abbildung 5.11 zeigt, werden Templates im Kompositionsmodell als „normale“ Komponenten repräsentiert und sind allein durch das boolesche Attribut `isTemplate` gekennzeichnet. Die Mächtigkeit der verwendeten Modellkonzepte entspricht – mit leichten Erweiterungen – der SMCDL (vgl. Abbildung 5.3). Insbesondere die semantische Typisierung der Schnittstelle über Modellreferenzen wird für Templates vorausgesetzt, während die Angabe implementierungsspezifischer Artefakte wie Vorbedingungen und Transformationsvorschriften entfallen kann.

Eine Besonderheit von Templates gegenüber der konventionellen Modellierung von Komponenten liegt in der optionalen *Gewichtung* von Modellreferenzen an Operationen und Ereignissen. In Anlehnung an die Zielbeschreibung aus WSMO kann die Sicht des Anwendungsentwicklers auf die Komponenten individuell abgebildet werden. Sind Modellreferenzen mit der Wichtung `1.0` belegt, was standardmäßig der Fall ist, so ist das Konzept von der gesuchten Komponente zu unterstützen. Wünschenswerte Eigenschaften können zusätzlich ausgedrückt werden – vergleichbar mit der *Intention* des Dienstnutzers nach Fensel et al. (2008), indem niedriger gewichtete semantische Referenzen angegeben werden.
Abbildung 5.11 veranschaulicht die semantische Typisierung in Form der Klasse ModelReference. Neben einer eindeutigen URI und der Gewichtung $weight \in [0, 1]$ enthält sie eine menschenlesbare Beschreibung des Konzeptes, die durch das Autorenwerkzeug aus dem Domänenmodell extrahiert werden kann.

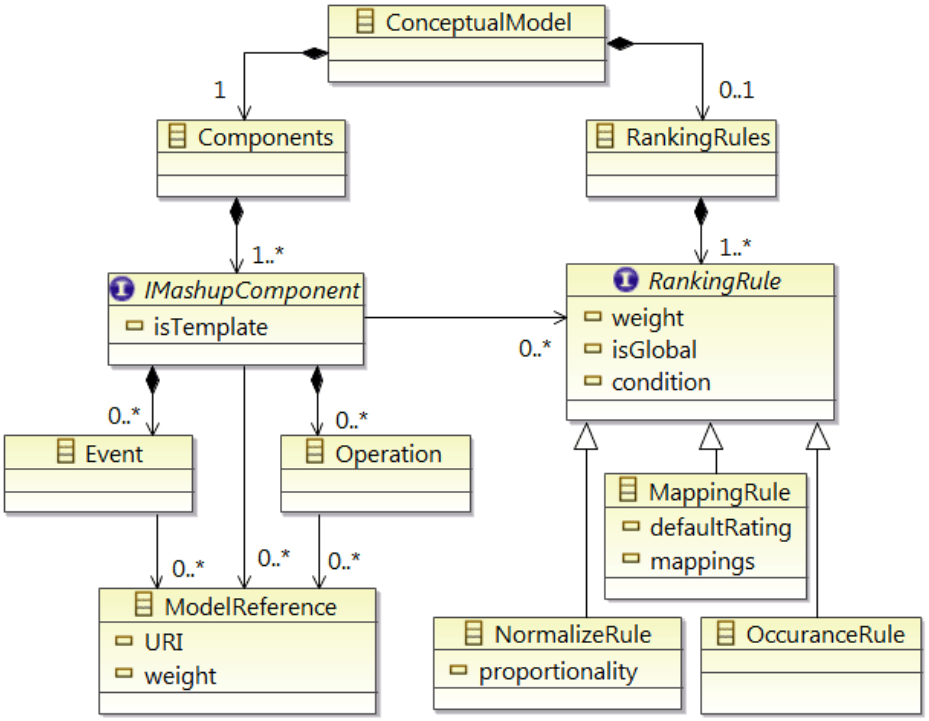

Abb. 5.11: Modellierung von Vorlagen als Teil des Kompositionsmodells

Optionale Operationen und Ereignisse werden für Templates explizit nicht unterstützt, da sie bei der dynamischen Integration von Komponenten zur Laufzeit zu Problemen führen können. Eine automatische Verknüpfung optionaler Bestandteile durch die Laufzeitumgebung kann allein schon aufgrund der fehlenden Eindeutigkeit nicht erfolgen. Auf der anderen Seite kann nicht ohne weiteres entschieden werden, ob das Fehlen vormodellierter Bestandteile und Verknüpfungen die Funktionsweise der Gesamtanwendung beeinträchtigt.
Templates stellen eine Teilmenge der SMCDL dar, da sie u. a. keine Metadaten aufweisen. Letztere sind nur Teil konkreter Instanzen und beeinflussen deren kontextsensitive Suche und Auswahl. Dazu kann jede Vorlage im Modell mit *Ranking-Regeln* verknüpft werden, die die Sortierung steuern. Sie werden im Container RankingRules hinterlegt (oder von extern verlinkt) und durch Vorlagen referenziert.

Neben der direkten Verknüpfung können Sortierungsregeln durch das Attribut `isGlobal` als anwendungsweit deklariert werden und somit für sämtliche Vorlagen gelten – beispielsweise, wenn eine Anwendung mitsamt allen Komponenten kostenfrei sein soll. Auch die unterschiedliche Gewichtung von Regeln kann über ihr Attribut `weight` ausgedrückt werden. Auf mögliche Ausprägungen der Ranking-Regeln und ihre Nutzung im Integrationsprozess wird in Abschnitt 6.1.3.1 eingegangen.

Fazit

Durch das Konzept der Templates wird dem Anwendungsentwickler die Abstraktion von konkreten Anwendungsbestandteilen ermöglicht, was zur Plattform- und Technologieunabhängigkeit des Kompositionsmodells beiträgt. Entwickler können selbst die Entscheidung treffen, ob sie Komponenten spezifisch oder abstrahiert im Modell repräsentiert sehen wollen.
Im ersten Fall erfolgt die Integration der über die ID referenzierten Komponentenimplementierung. Aufgrund der gleichartigen Modellierung von konkreten und abstrakten Komponenten kann zur Laufzeit jede Komponente auch als Template verstanden werden. Dazu werden alle bestehenden Modellreferenzen als obligatorisch betrachtet, und die anwendungsweiten Ranking-Regeln als Grundlage zur Sortierung genutzt. Dies ermöglicht die Portabilität von Anwendungen auf andere Ausführungsplattformen und erlaubt den kontextabhängigen oder nutzergetriebenen Komponententausch, falls erwünscht.
Im Fall der Abstraktion wird zur Laufzeit durch die Kompositionsumgebung eine semantisch passende Komponente gesucht. Diese muss nicht zwangsläufig exakt der Vorlage entsprechen – sie kann beispielsweise mehr Operationen und Ereignisse aufweisen. Zudem kann durch die gewichteten Modellreferenzen eine Eingrenzung der Kandidatenmenge erreicht werden. Neben der funktionalen Passgenauigkeit spielt für die Auswahl insbesondere der Kontext der Nutzung eine Rolle, wie Nutzer- und Endgeräteeigenschaften. Der Integrationsprozess, insbesondere *Matching* und *Ranking* auf Basis von Templates, wird ausführlich in Abschnitt 6.1 erörtert.
Der Daten- und Kontrollfluss zwischen Komponenten beruht auf Ereignissen und Operationen, welche Nachrichten mit typisierten Parametern senden und verarbeiten. In der einfachsten, rein syntaktischen Form des Kompositionsmodells können die entsprechenden Datentypen direkt aus den Komponentenbeschreibungen extrahiert, unter DataTypes abgelegt und von Parametern und Eigenschaften referenziert werden. Vorzuziehen ist die semantische Typisierung aller Daten einer Anwendung durch Referenzen auf externe Domänenmodelle, wie bei der Komponentenbeschreibung diskutiert. In diesem Fall entfällt der Datentyp-Bereich komplett und die Integration semantisch kompatibler Komponenten wird ermöglicht.

5.2.1.2 Styles

Das Teilmodell der *Styles* kann genutzt werden, um visuelle Eigenschaften entkoppelt und wiederverwendbar zur modellieren. Ihr Zweck und Einsatz ähnelt dem Ansatz von Klassen in CSS: Für verschiedenste audio-visuelle Eigenschaften wie *Lautstärke*, *Schriftart*, *-größe* und *-farbe* stehen Modellklassen und -attribute zur Verfügung die definiert bzw. instanziiert und verschiedenen Komponenten zugeordnet werden können. Dadurch wird ein homogenes Erscheinungsbild der Gesamtanwendung

sichergestellt, was freilich die Unterstützung durch die jeweiligen Komponenten voraussetzt. Wie bei CSS erlaubt das Modell die Kaskadierung von Stilen.
Die Modellierung und Anwendung von Stilen wird in Abbildung 5.12 exemplarisch veranschaulicht. Der Stil *AppStyleCD* fasst drei Stildefinitionen zusammen (*contains*), die das Corporate Design hinsichtlich der Formatierung von Kanten, Listen und Schriftarten repräsentieren. Die Anwendung von Stilen auf UI-Komponenten wird durch die Referenzierung über *decorates* im Modell ausgedrückt.

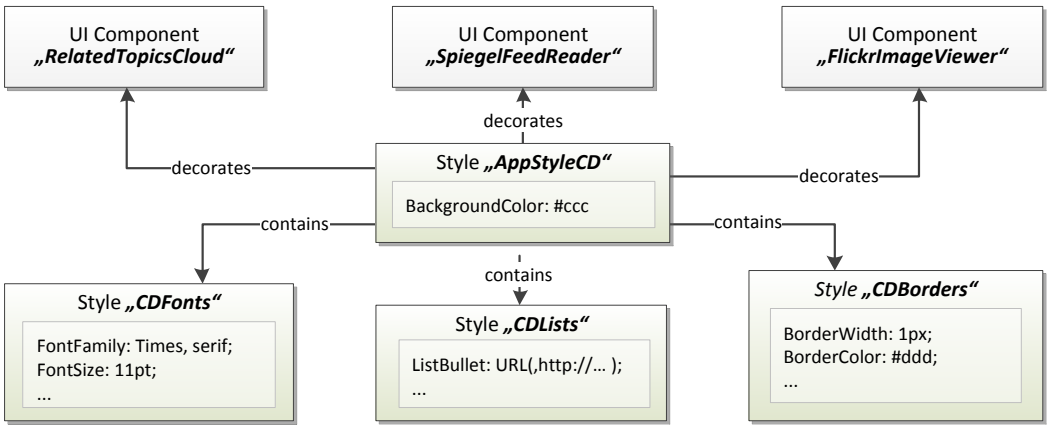

Abb. 5.12: Anwendung von Stilen auf UI-Komponenten am Beispiel

Bei der Transformation des Kompositionsmodells oder dessen Interpretation durch die Laufzeitumgebung muss die Abbildung der Styles in Objekte der Plattform erfolgen. Diese werden Komponenten bei ihrer Initialisierung zur Verfügung gestellt (vgl. Abschnitt 6.2.3.1) und müssen intern umgesetzt werden. In den umgesetzten Prototypen konnte die Abbildung in CSS erfolgen, da dies sowohl durch client- als auch durch die serverseitige Plattform (Eclipse RAP) unterstützt wurde.

5.2.1.3 Laufzeitaspekte

Der letzte Bestandteil des Conceptual Models ist das *Environment*, welches alle externen Aspekte einer Komposition kapselt und zwei Teilbereiche umfasst. Innerhalb von PublicInterface können beliebige Ereignisse und Operationen innerhalb des Modells als *öffentlich* deklariert werden, d. h. sie werden für umgebende Kompositionen oder Rahmenanwendungen nutzbar und formen die äußere Schnittstelle der entstehenden kompositen Komponente. Die Klasse Runtime repräsentiert die Laufzeitaspekte der Komposition. Über sie ist es u. a. möglich, anwendungsweite Variablen und sog. InitEvents zu definieren. Letztere stellen Ereignisse dar, die beim Anwendungsstart ausgelöst werden. Die damit verbundenen Operationen werden bei der Initialisierung der Anwendung aufgerufen und lösen den gewünschten Effekt aus, z. B. das Laden oder die Vorselektion bestimmter Daten.

Über das einheitliche Komponentenmodell sind alle Anwendungsbestandteile mitsamt der Schnittstelle zu externen Systemen nach den gleichen Prinzipien im Modell repräsentiert. Der nächste Schritt zu einer funktionsfähigen Mashup-Anwendung ist nun die Verknüpfung der Komponenten, um den Daten- und Kontrollfluss der Anwendung zu beschreiben. Diesem Aspekt widmet sich der folgende Abschnitt.

5.2.2 *Communication Model* – Spezifikation von Daten- und Kontrollfluss

Nach Erl (2005) ist die Anwendungslogik im Kontext von SOA auf zwei Ebenen verteilt: Auf der Schnittstellenebene kommunizieren lose gekoppelte Dienste über offene Protokolle miteinander, während auf der Anwendungsebene Logik „im Verborgenen“, d. h. innerhalb der Dienste, verschiedenartig umgesetzt ist. Diese Trennung findet sich gleichsam in Mashups wieder. Die Kopplung von Komponenten über das Communication Model entspricht der Verknüpfung auf der Schnittstellenebene, während auf die Interna der Komponenten kein Einfluss genommen werden kann. Durch die Verknüpfung kann der Daten- und Kontrollfluss einer kompositen Anwendung beschrieben werden. Komponenten – Datenquellen gleichsam wie UI-Bestandteile – können miteinander synchronisiert bzw. integriert werden, wodurch sich die Anwendung von klassischen Portalen und *Dashboards* abhebt.
Im vorliegenden Modell wird dazu ein datenflussorientiertes Paradigma genutzt, d. h. es wird kein expliziter Kontrollfluss zwischen den Komponenten modelliert, da der Zielgruppe der Nicht-Programmierer kein Wissen über Kontrollflüsse und die nötigen Programmkonstrukte unterstellt werden kann. Der Schwerpunkt der Modellierung widmet sich der Frage, wie sich Daten durch die Anwendung bewegen bzw. zwischen Komponenten ausgetauscht werden. Durch die Weiterleitung von Daten (und auch potentiell leerer Nachrichten) ergibt sich ein impliziter Kontrollfluss.

Die Modellierung der Kommunikation richtet sich – unter Beachtung der losen Kopplung aller Anwendungsbestandteile – an den Möglichkeiten des Komponentenmodells aus, die ausführlich in Abschnitt 5.1.1 beschrieben wurden. Zur Kopplung von Komponenten stehen folglich Properties, Events und Operations zur Verfügung. Die dadurch implizierte ereignisgesteuerte Kommunikation zählt zu den meist verwendeten im Softwareumfeld [Faison, 2006]. Ereignisse signalisieren eine Zustandsänderung in einem System bzw. einer Komponente und führen zur Publikation entsprechender Nachrichten nach außen. Alle interessierten Empfänger können diese verarbeiten. Zur Verteilung der Nachrichten wird das *Publish/Subscribe*-Prinzip genutzt ist. Dessen großer Vorteil gegenüber dem verbreiteten Request-Reponse-Paradigma liegt in der losen Kopplung der Kommunikationspartner: Da die Kommunikation über einen Vermittler abläuft, müssen sich Auslöser von Ereignissen und deren Empfänger nicht kennen.

5.2.2.1 Verknüpfung durch Links

Im Kommunikationsmodell übernehmen *Links* (im Weiteren auch als *Kanäle* bezeichnet) die Rolle dieser Vermittler. Ein Link verknüpft potentiell n Events mit m Operations, sofern ihre Parameter kompatibel, d. h. hinsichtlich ihrer Datensemantik aufeinander abbildbar sind. Dabei bleiben die Komponenten lose gekoppelt – sie sollen und dürfen sich untereinander nicht kennen, sondern kommunizieren ausschließlich über Links der Anwendung. Um die vielfältigen Anforderungen an den Kontroll- und Datenfluss in interaktiven Anwendungen adäquat auf die ereignisorientierte Kommunikation abzubilden – in Abschnitt 2.3 wurden diesbezüglich diverse Anforderungen aufgestellt – werden drei Arten von Links unterschieden:

Link Links verknüpfen n *Publisher* (Ereignisse) mit m *Subscribern* (Operationen). Abbildung 5.13 veranschaulicht die Verbindung schematisch. Die Daten eines

am Link anliegenden Events (in der Abbildung ein Parameter vom Typ X) werden an alle registrierten Operationen publiziert, wobei keinerlei Rückantwort oder Bestätigung erwartet wird. Somit kann eine einfache **Fire-and-Forget**-Kommunikation modelliert werden. Jeder Link ist implizit durch die Signatur der übertragenen Daten typisiert. Somit können nur diejenigen Ereignisse und Operationen mit dem Link und somit miteinander verbunden werden, deren Parameter hinsichtlich Anzahl und Datentyp übereinstimmen.

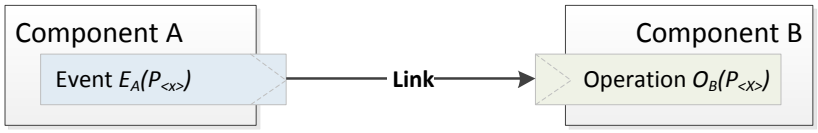

Abb. 5.13: Einfache Verknüpfung von Komponenten über Links

BackLink Über einen BackLink wird die Zwei-Wege-Kommunikation zwischen Komponenten ermöglicht, die in ereignisbasierten Systemen sonst nur über Umwege (und nur auf Basis impliziten Wissens) modelliert werden kann. BackLinks verbinden n *Requestor* (Events) mit genau einem *Replier* (Operation), wie Abbildung 5.14 zeigt. Im Gegensatz zum Link wird dadurch zur Laufzeit ein temporärer Rückkanal aufgebaut, über den nach dem initialen Aufruf eine Antwort erfolgen kann. Dazu muss das Ereignis auf eine Rückruf-Operation $O_A(P_{<Y>})$ verweisen (*callbackOperation*, vgl. Abschnitt 5.1.3.2), während die aufgerufene Operation wiederum ein Rückgabe-Event referenzieren muss (*callbackEvent*, ebd.). Sobald ein Event auf dem Kanal publiziert wird, erfolgt der Aufruf der entsprechenden Ziel-Operation und eine Punkt-zu-Punkt-Verbindung zwischen dem Rückgabe-Ereignis und der Rückruf-Operation wird aufgebaut. Über diesen Kanal kann die Antwort, z. B. in Form eines Rückgabewertes, einer Erfolgs- oder Fehlermeldung, erfolgen, womit das **Request-Response**-Paradigma auf die ereignisorientierte Kommunikation abgebildet wurde.

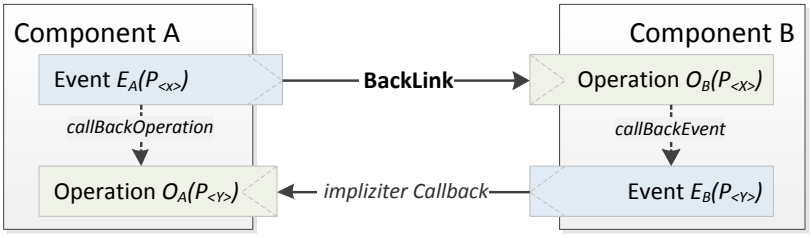

Abb. 5.14: Verknüpfung mit Rückkanal über BackLinks

Die Entscheidung, ob dieser implizite Rückkanal nach einmaliger Rückgabe aufrechterhalten wird, kann bei der Modellierung getroffen werden. Neben der einmaligen Anfrage (*single pull*) ist es möglich, dauerhafte Aktualisierungen (*active push*) von Daten zu erlauben. Ist der Kanal entsprechend markiert, kann die Empfängerkomponente wiederholt Daten zur Rückrufoperation senden. Dieses Verhalten muss explizit unterstützt werden, was durch das MCDL-Attribut `syncable` der angefragten Operation angezeigt wird. Durch Angabe eines `syncThresholds` als Teil des BackLinks kann ein Schwellwert für die Datenübermittlung, d. h. die minimale Zeit zwischen zwei Aktualisierungen,

angegeben werden, um negativem Einfluss auf die Performanz vorzubeugen. Häufiger Anwendungsfall für eine solche Kommunikation ist die Abfrage veränderlicher Daten, wie die Börsenkurse im Beispielszenario, durch visualisierende Komponenten. Statt dem wiederholten *Polling* der Daten von entsprechenden Dienst-Komponenten reicht die Modellierung eines einzigen, permanenten BackLinks, um Komponenten permanent und aktiv mit Daten zu versorgen.

PropertyLink Ein PropertyLink erlaubt die **Synchronisation** von Komponentenzuständen über deren zustandsgebende Eigenschaften. Ein häufiges Einsatzfeld hierfür ist die visuelle Datenanalyse, bei der Komponenten verschiedene Sichten auf eine Datenmenge erlauben. PropertyLinks können diese Sichten konsistent halten, z. B. bezüglich eines ausgewählten Zeitraumes wie im Aktien-Szenario. Die Synchronisation der Properties erfolgt implizit durch die entsprechenden Change-Events und Setter-Methoden, wie in Abbildung 5.15 veranschaulicht. Beide sind für alle deklarierten Eigenschaften verpflichtend umzusetzen, müssen aber nicht explizit in der Schnittstellenbeschreibung deklariert werden. Da das Kommunikationsmuster eine wechselseitige Synchronisation vorsieht, werden die Teilnehmer einheitlich als SyncTarget modelliert, von denen jeder PropertyLink mindestens zwei besitzt.

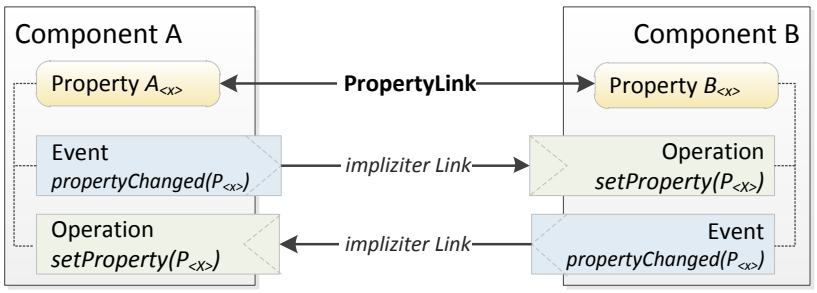

Abb. 5.15: Synchronisation von Eigenschaften über PropertyLinks

Der PropertyLink kann letztlich als zusätzliche Abstraktionsstufe im Modell betrachtet werden. Gegenüber Links zwischen den impliziten Ereignissen und Operationen wird die Modellierung stark vereinfacht und Redundanz im Modell vermieden, falls Synchronisation zwischen mehreren Komponenten gewünscht wird. Zudem erlaubt die explizite Modellierung den Laufzeitumgebungen eine gesonderte Behandlung der Synchronisationslinks. Dies ist nötig, um bei der konventionellen Modellierung entstehende Zyklen („*A benachrichtigt B benachrichtigt A ...*“) unterbinden zu können.

Abbildung 5.16 zeigt die wichtigsten Klassen des Kommunikationsmodells im Überblick. An oberster Stelle ist die Unterscheidung der drei Linkarten – PropertyLink, Link und BackLink erkennbar. Erstere verbinden mindestens zwei Properties (referenziert durch die Klasse SyncTarget) miteinander. Die beiden anderen Link-Klassen definieren die bereits benannten Sender und Empfänger von Nachrichten, die jeweils auf Ereignisse oder Operationen von Komponenten im Conceptual Model verweisen. Die Kompatibilität angeschlossener Events und Operations wird dadurch gewährleistet, dass jeder Link eine eigene Signatur, d. h. eine feste Anzahl und Reihenfolge typisierter Parameter, besitzt. Ein Link erbt somit gleichsam wie Events

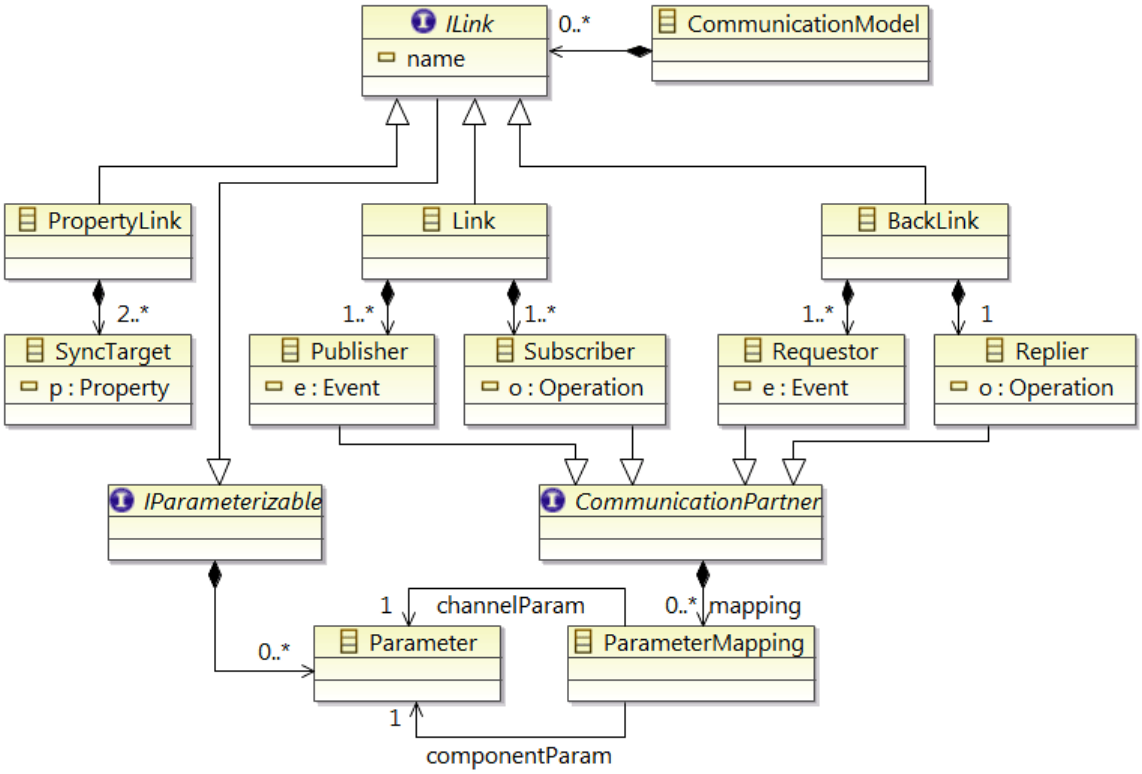

Abb. 5.16: Modellierung von Daten- und Kontrollfluss im *Communication Model*

und Operations von IParameterizable und setzt die semantische Kompatibilität der Signaturen von Sendern und Empfängern voraus, die im Modell durch Object Constraint Language (OCL)-Constraints sichergestellt wird. Auf die manuelle Abbildung über ParameterMappings wird im folgenden Abschnitt eingegangen.

Tabelle 5.1 fasst die wichtigsten Eigenschaften der vorgestellten Konzepte zusammen. Über Links können beliebig viele Ereignisse und Operationen verknüpft, und über PropertyLinks beliebig viele Properties miteinander synchronisiert werden. Während ein BackLink im Modell n Events mit genau einer Operation (mit Rückgabewert) verbindet, so findet zur Laufzeit eine 1:1-Kommunikation statt, d. h. nur die Komponente mit dem auslösenden Event erhält den Rückgabewert der aufgerufenen Operationen über deren Rückgabe-Event. Da Ereignisse nicht zwingend Parameter enthalten, lässt sich über Links zudem der Kontrollfluss einer Anwendung modellieren. Sie können u. a. im Rahmen des *Life-Cycle-Managements* verwendet werden und die Integration, die Initialisierung oder Fehler anzeigen.

Während der einfache Link die *Fire-and-Forget*-Metapher (*Push*) umsetzt, wie sie die Mehrzahl existierender Systeme unterstützt, kann durch den Rückkanal von BackLinks auch das *Request-Response*-Muster realisiert werden, um aktiv Daten abzufragen (*Pull*). Die Synchronisation über PropertyLinks entspricht schließlich einem impliziten *Push* durch Komponenten auf einem gemeinsamen Kanal. Die Übermittlung von Nachrichten per Events über Links und PropertyLinks ist atomar und läuft somit zeitlich synchron ab. BackLinks sind hingegen zeitlich asynchron einzuschätzen, da die Rückgabe nicht durch die aufgerufene Operation, sondern entkoppelt davon durch das Rückgabe-Event erfolgt.

	Link	BackLink	PropertyLink
Kardinalität	m:n	1:1	m:n
Programmfluss	Daten- und Kontrollfluss	Daten- und Kontrollfluss	Datenfluss
Datenfluss-Richtung	Push	Push und Pull	Impliziter Push
Intention	Publikation (*Fire and Forget*)	Publikation mit Rückmeldung (*Request-Reponse*)	Synchronisation
Zeitliche Verarbeitung	Synchron	Asynchron	Synchron

Tab. 5.1: Gegenüberstellung der Linkformen im Communication Model

Somit können alle nötigen Kommunikationsmuster für interaktive komposite Webanwendungen repräsentiert werden, die in Abschnitt 2.3 gefordert wurden. Durch die Verbindung über Links bleibt die lose Kopplung zwischen Komponenten erhalten. Die Kommunikation ist jedoch nicht auf die Weiterleitung von Daten beschränkt, wie in fast allen existierenden Mashup-Systemen, sondern ermöglicht auch die Abfrage von Daten sowie die dauerhafte Aktualisierung über einmal geöffnete BackLinks. PropertyLinks erlauben schließlich die Synchronisation von Komponentenzuständen und stellen eine zusätzliche Abstraktionsstufe des Kommunikationsmodells dar.

Einige Beispiele zur praktischen Anwendung der Modellierungsmittel finden sich in Abschnitt 7.3 bei der Diskussion der Beispielanwendungen.

5.2.2.2 Steigerung der Interoperabilität

Da die Entwicklung von Komponenten potentiell unabhängig voneinander erfolgt, kann die Passgenauigkeit von Ereignissen und Operationen in der Praxis nicht unterstellt werden. Die Kompatibilität von Komponentenschnittstellen kann sowohl auf Signatur- als auch auf Datenebene beeinträchtigt sein.

Abweichende Signaturen zwischen Events und Operations äußern sich in der unterschiedlichen Anzahl oder Reihenfolge der enthaltenen Parameter. Ein Event E_1 mit der Signatur $\{\textit{EventName}, \textit{Location}\}$ kann nicht ohne weiteres mit einer Operation der Signatur $\{\textit{Location}\}$ verknüpft werden. Probleme dieser Art werden im Kompositionsmodell in Form von ParameterMappings gelöst, welche einzelne Parameter der Kommunikationspartner auf die Signatur des Kommunikationskanals abbilden (vgl. Abbildung 5.16). Dabei können Parameter beliebig umsortiert oder verworfen werden.

Beispiel: *Abbildung 5.17 zeigt den Einsatz von ParameterMappings am Beispiel von drei Komponenten. Der Kommunikationskanal bzw. Link ist durch zwei darüber ausgetauschte Parameter* $LastName$ *und* Age *definiert. Der nicht benötigte Parameter* ID *des Events* $customerSelected$ *von Komponente* B *kann beim ParameterMapping entfallen. Die beiden anderen Parameter können direkt auf die Link-Parameter abgebildet werden. Auf der Empfängerseite wird für den Aufruf von* $addCandidate$ *die umgedrehte Reihenfolge der Parameter benötigt. Auch diese Umsortierung erfolgt durch die Zuordnung von Link- zu Operationsparametern über ein ParameterMapping.*

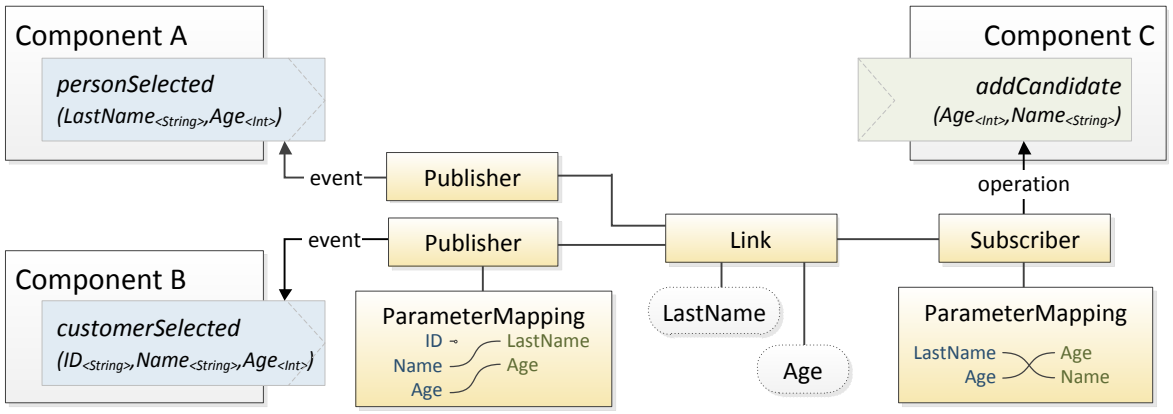

Abb. 5.17: Exemplarisches ParameterMapping

Der Einsatz von ParameterMappings stellt für die Verknüpfung von Komponenten mit abweichenden Signaturen eine enorme Vereinfachung dar. Er ist jedoch nur dann nötig, falls Parameter tatsächlich entfallen oder umsortiert werden müssen. Sind keine Mappings definiert, so werden die Parameter entsprechend ihrer Reihenfolge in der Komponentenbeschreibung weitergeleitet. Neben der manuellen Anpassung, die nur durch Entwickler möglich wäre, könnte die Abbildung auch mit den bereits angesprochenen Logikkomponenten (Abschnitt 5.2.1) erfolgen. Die Einbindung dieser ist jedoch in diesem Fall nicht praktikabel. Die Umsortierung von Parametern, wie im Beispiel, würde zunächst die Aufsplittung (EventSplit) und danach die Zusammenführung der Event-Parameter (EventJoin) nötig machen, was den Umfang und die Komplexität des Modells zu Lasten der Einfachheit und Verständlichkeit steigern würde. Durch den Einsatz von ParameterMappings können Signaturunterschiede deutlich einfacher und verständlicher überbrückt werden.

Inkompatibilitäten auf Datenebene treten auf, wenn Parameter zwar dem gleichen semantischen Typ entsprechen, syntaktisch jedoch unterschiedlich manifestiert sind. Um diese zu lösen, erfolgt die Abbildung komponenteninterner Datentypen auf die konzeptionell vorgegebenen *Groundings* für semantische Typen (vgl. Abschnitt 5.1.3.3) bereits vor der Publikation auf die Kommunikationskanäle. Dazu kommen (falls nötig) die Transformationen aus den Bindings der Komponentenbeschreibungen (vgl. Abschnitt 5.1.3.2) zum Einsatz. Doch auch die Nutzung einheitlicher Groundings impliziert bewusst keine Typgleichheit semantisch kompatibler Daten auf syntaktischer Ebene, da sie sich auf unterschiedlichen Ebenen der Vererbungshierarchie befinden können.

Beispiel: *Ein Event E_1 stellt ein Datum vom Typ EventLocation bereit, während die damit verknüpfte Operation einen Parameter vom Typ Location erwartet. Beide können miteinander verbunden werden, sofern eine Subklassenbeziehung zwischen den Typen besteht. In diesem Fall sind alle nötigen Informationen in E_1 vorhanden, es muss jedoch ein „Cast“ des Datums entlang der Vererbungshierarchie erfolgen.*

Derartige Abbildungen werden zur Laufzeit durch Mediationsmechanismen sichergestellt, die in Abschnitt 6.2.4.2 erläutert werden.

5.2.2.3 Modellierung von Drag-and-Drop-Interaktionen

Eine weitere Anforderung hinsichtlich der Kommunikation besteht in der Unterstützung plattformspezifischer Interaktionsmuster, wie *Drag-and-Drop*, Gesten und Spracheingabe. Prinzipiell ist es Aufgabe der Laufzeitumgebungen, diese auf das Modell abzubilden bzw. umgekehrt. Die Drag-and-Drop-Interaktion bildet jedoch einen Sonderfall, da sie eine *on-demand*-Verknüpfung durch den Nutzer darstellt, die nicht permanent im Kommunikationsmodell besteht. Wie am Beispiel der Aktienverwaltung in Abschnitt 2.2.2 gezeigt wurde, liegt darin der Mehrwert: Nicht eindeutige Verknüpfungen werden durch Nutzer selbst hergestellt (im Beispiel: *„In welcher Detailansicht soll der ausgewählte Aktienkurs visualisiert werden?“*). Nach der Datenübermittlung verschwindet der temporäre Kommunikationskanal.
Um diese ad-hoc-Verknüpfung zu unterstützen, müssen Komponenten darüber Auskunft geben, welche Informationen sie über *Drag* bereitstellen und über *Drop* verarbeiten können. Den Ausgangspunkt der Interaktion zeigt das MCDL-Element `dataSource` an, welches die bewegten Daten in Form typisierter Parameter enthält. Im Kompositionsmodell wird es ebenfalls als Teil der Komponentenschnittstelle repräsentiert wird. Die potentiellen Ziele der Interaktion sind bereits implizit im Modell vorhanden: Operationen zeigen an, welche Daten eine Komponente verarbeiten kann – unabhängig davon, wie sie aufgerufen werden. Die Vermittlung zwischen Auslösern und Zielen fällt in den Verantwortungsbereich der Laufzeitumgebung und wird in Abschnitt 6.2.4 erläutert. Aus Modellierungssicht stellt sich jedoch die Frage, inwiefern der Kompositeur auf diese Form der Interaktion Einfluss nehmen kann.
Werden zur Laufzeit Daten auf eine Komponente gezogen, deren Signatur (Anzahl und Typ der übermittelten Parameter) mit denen einer Operation kompatibel sind, so wird letztere mit den übermittelten Daten aufgerufen. Dieses Verhalten kann durch das Attribut `excludeDrop` von Operationen beeinflusst werden. Durch Setzen des Wertes *true* kann die entsprechende Operation nicht mehr aufgerufen werden. Das Attribut existiert gleichsam für UI-Komponenten, falls diese gänzlich von der Drag-and-Drop-Interaktion ausgeschlossen werden sollen.

In seiner Gesamtheit bietet das Kommunikationsmodell Entwicklern vielfältige Möglichkeiten, die verschiedenen nötigen Kommunikations- und Koordinationsformen interaktiver Anwendung auf eine Komposition lose gekoppelter Bestandteile anzuwenden. Das Modell repräsentiert die ereignisorientierte Kommunikation zwischen Komponenten und ist in erster Linie datenflussorientiert. Da Events jedoch nicht zwangsläufig Daten beinhalten bzw. transportieren müssen, wird implizit auch der Kontrollfluss der Anwendung beschrieben. Neben der weit verbreiteten unidirektionalen Verknüpfung über Events können auch bidirektionale Request-Reponse- und Synchronisationsbeziehungen zwischen Komponenten modelliert werden. Von plattformspezifischen Interaktionstechniken wird dabei abstrahiert. Allein das Verhalten bei der Drag-and-Drop-Interaktion kann, aufgrund seiner besonderen Rolle, durch den Entwickler im Modell beeinflusst werden. Zur Steigerung der Interoperabilität bietet das Kommunikationsmodell insbesondere Konstrukte, um die Abweichungen auf der Signaturebene von Operationen und Ereignissen einfach auflösen zu können. Die Behandlung syntaktischer Differenzen der Datentypen erfolgt – semantische Kompatibilität vorausgesetzt – zur Laufzeit im Rahmen der Mediation und muss nicht explizit modelliert werden.

5.2.3 *Layout Model* – Visuelle Anordnung von UI-Komponenten

Die Anordnung von UI-Komponenten auf der Oberfläche wird durch das Layout Model beschrieben. Dieses kann mit Instanzen vordefinierter Layout-Klassen gefüllt werden, die Autoren aus gängigen Frameworks, wie Java Swing und SWT bekannt sind und die Anordnung pixelgenau positioniert (AbsoluteLayout), in Form von Tabs (TabLayout) in Gitterform (GridLayout) oder nacheinander aufgereiht in einer Box (FillLayout) ermöglichen. Das Attribut `agile` signalisiert, ob das Layout durch den Nutzer dynamisch verändert werden kann, z. B. durch Verschieben der Komponenten. Abbildung 5.18 zeigt einen Ausschnitt des Teilmodells auf der linken Seite. Es wird ersichtlich, dass jedes Layout auf der untersten Ebene LayoutElements enthält, die Tabellenzellen, Positionen oder Tabs repräsentieren. Sie verweisen entweder auf eine UI-Komponente des Conceptual Models, die an der entsprechenden Stelle gerendert werden soll, oder sie enthalten ein weiteres Layout (*sublayout*). Durch die Möglichkeit der hierarchischen Schachtelung können beliebige Anordnungen realisiert werden. Das Aussehen der Layout-Container, z. B. deren Rahmen und Hintergrundfarben, kann durch die o. g. visuellen Stile beeinflusst werden, um ein konsistentes visuelles Erscheinungsbild der Gesamtanwendung sicherzustellen.
Das Layout Model unterstützt die folgenden Layouts:

GridLayout Dieses Layout eignet sich zur Positionierung von UI-Komponenten in einem Gitternetz. Es erlaubt die Definition von Zeilen (*Rows*) oder Spalten (*Columns*), die sich auf unterster Ebene aus mehreren Gitterzellen (*Cells*) zusammensetzen. Jede Zelle entspricht einem *LayoutElement* und kann entsprechend weitere Layouts oder einen Verweis auf eine darzustellende Komponente enthalten.

FillLayout Abgeleitet vom bekannten Fill- bzw. FlowLayout ermöglicht diese Klasse, Komponenten horizontal oder vertikal anzuordnen, wobei die Ausrichtung durch das Attribut `fillStyle` angezeigt wird. Der Zeilen/Spaltenumbruch erfolgt automatisch durch die Laufzeitumgebung, falls erforderlich.

TabLayout Bei diesem Layout werden alle LayoutElements als separate Tabs angezeigt. Insbesondere beim TabLayout spielt die Anwendung von Styles aus dem Conceptual Model eine große Rolle, da es im Gegensatz zu den restlichen Layouts über eigene visuelle Artefakte verfügt.

AbsoluteLayout Durch die Definition von Koordinaten können LayoutElements pixelgenau auf der Oberfläche positioniert werden. Somit kann auch die Überlappung visueller Komponenten erreicht werden, falls gewünscht.

Abbildung 5.19 zeigt die Anwendung der Konzepte am Beispiel des *TravelMash*-Szenarios aus Abschnitt 2.2.1. Die Oberfläche ist auf oberster Ebene durch ein GridLayout in drei Zeilen unterteilt. Jede Zeile enthält eine oder mehrere Zellen bzw. LayoutElements (s. o.). Diese können mit unterschiedlichen Breiten konfiguriert werden und enthalten i. d. R. UI-Komponenten, z. B. zur Auswahl der Orte und zur Auflistung von Hotel- und Event-Informationen. Die erste Zelle der ersten Zeile veranschaulicht die Hierarchisierung anhand des enthaltenen FillLayouts, welches zur vertikalen Anordnung der Kalender- und Wetter-Komponenten dient.
Die Modellierung der räumlichen Anordnung aller sichtbaren Komponenten ist einfach, aber effektiv. Durch die Kombination und Schachtelung grundlegender

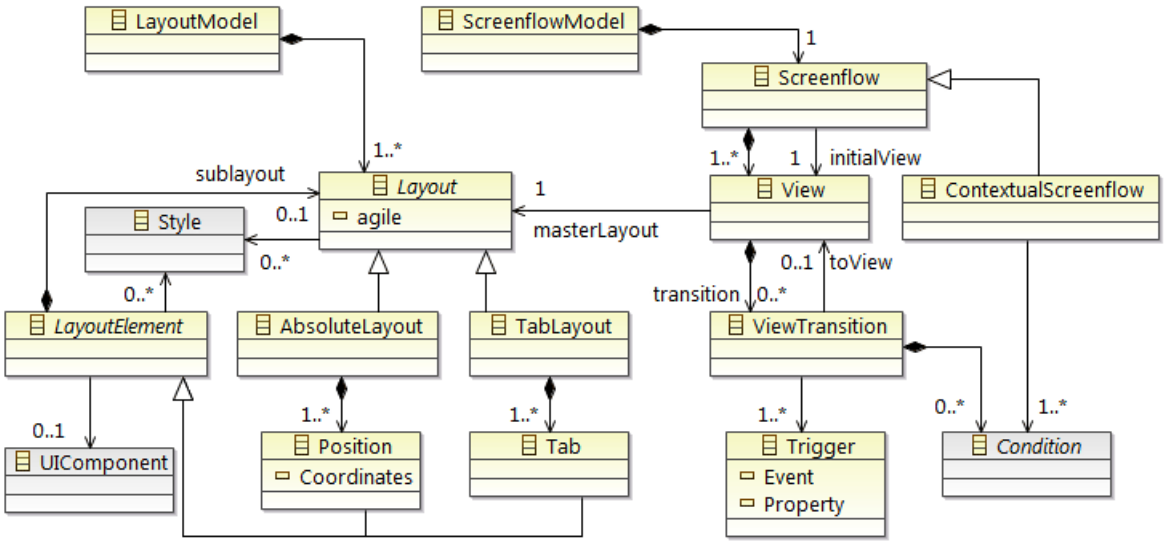

Abb. 5.18: Definition von Layouts und Sichten im Kompositionsmodell

Layouts können beliebige Anordnungen erreicht werden. Gleichzeitig bleiben die Komponenten selbst unabhängig vom Layoutmodell, da sie nur referenziert werden. Entwickler sind nicht auf ein *Master-Layout* beschränkt, sondern können mehrere, ggf. alternative Layouts im Modell definieren. Diese bilden die Grundlage für verschiedene Sichten auf die Anwendung, z. B. je nach Nutzergruppe oder Endgeräteklasse, deren Modellierung im nächsten Abschnitt behandelt wird.

5.2.4 *Screenflow Model* – Definition von Navigation und Sichten

Ziel des Screenflow Models ist es, die Navigation innerhalb einer kompositen Anwendung auf Basis der gegebenen, ereignisorientierte Kommunikation zu modellieren. Dadurch wird es möglich, je nach Anwendungszustand, Nutzer- oder Endgerätekontext, verschiedene Sichten auf die Anwendung und ihre Daten zu definieren. Dies ist insbesondere dann wichtig, wenn durch die Anwendung Workflows bzw. mehrstufige Prozesse abgebildet werden sollen, die in ihrer Gänze für den Nutzer nicht überschaubar und nur schwer auf einem Bildschirm darstellbar sind. Auf die Rolle der ContextualScreenflows wird im Rahmen der Adaptionsmechanismen in Abschnitt 5.3.2.2 eingegangen.

Jeder Screenflow besteht aus Sichten, die durch Instanzen der Klasse View repräsentiert werden (vgl. Abbildung 5.18, rechts). Diese entsprechen dem Konzept der Webseiten bzw. „pages“ in klassischen Modellierungsansätzen. In RIAs ist der Seitenbegriff weniger treffend, da kein Wechsel zwischen XHTML-Seiten mehr vorliegt, sondern Komponenten dynamisch hinzugefügt und entfernt oder nur ein- und ausgeblendet werden. Der Nutzer hat es mit einem visuellen Zustand der Anwendung zu tun, der durch die sichtbaren Komponenten auf der Oberfläche charakterisiert ist. Folglich referenziert jede Sicht ein Layout des Layout-Modells, welches gleichzeitig die anzuzeigenden Komponenten referenziert und ihre Anordnung beschreibt.

Eine Sicht muss als *initialView* deklariert werden, sodass sie beim Anwendungsstart geladen wird. Alle weiteren Sichten können von ihr aus durch Transitionen erreicht werden. Diese sind durch Instanzen der Klasse ViewTransition beschrieben, welche

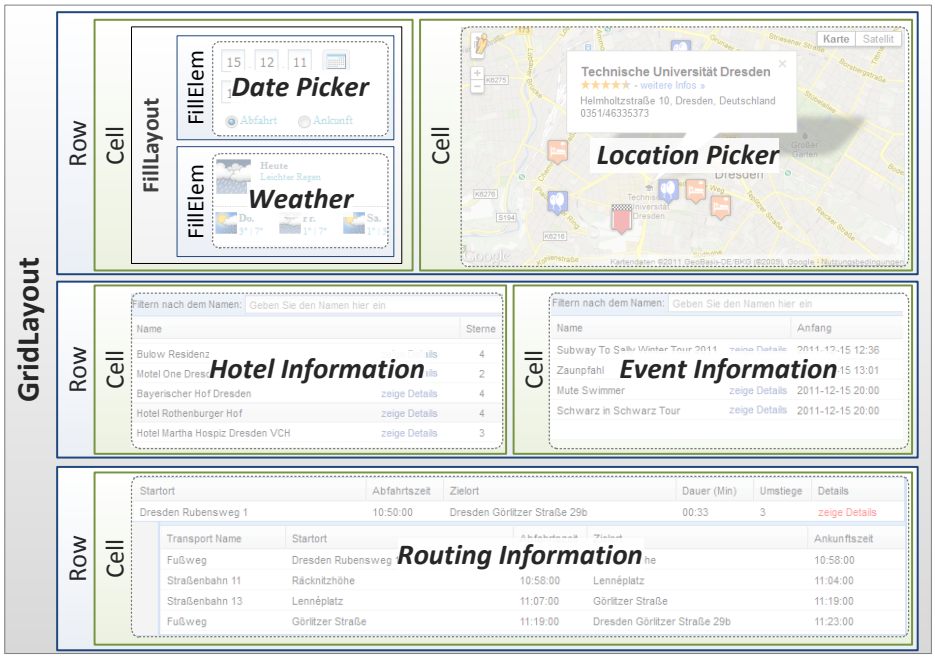

Abb. 5.19: Modellierung des Layouts am Beispiel

den Wechsel zwischen zwei Views repräsentiert. Ein solcher Wechsel wird durch Trigger ausgelöst, die explizite Events einer Komponente oder implizite Events aufgrund von Property-Änderungen darstellen. Durch die konsequente Nutzung des Komponentenmodells können diese Events gleichermaßen Zustandsänderungen von Komponenten, Systemereignisse der Plattform oder Kontextänderungen anzeigen. Ein Sichtwechsel kann demzufolge durch Nutzerinteraktion, Anwendungslogik, durch eine Rahmenanwendung oder die Kompositionsumgebung ausgelöst werden. Die universelle Nutzung von Events für den Daten- und Kontrollfluss bringt einen weiteren Vorteil mit sich: Daten können – im Gegensatz zur Mehrzahl existierender Modelle aus dem Web-Engineering-Umfeld, wie WebML – zwischen Sichten weitergegeben werden. Dazu müssen die Komponenten potentiell verschiedener Sichten lediglich im Kommunikationsmodell miteinander verbunden werden, und das auslösende Event durch eine ViewTransition referenziert werden. Gleichzeitig kann die Rolle als Publisher oder Subscriber an einem Kommunikationskanal (vgl. Abschnitt 5.2.2.1) durch Querreferenzen auf bestimmte Views beschränkt sein, um den Kommunikationsoverhead zu minimieren.

Die Ausführung von Transitionen kann an Bedingungen geknüpft werden, die im Modell durch die abstrakte Klasse Condition repräsentiert werden. Die Bedingungslogik wird durch Klassen beschrieben, die u. a. die boolesche Verknüpfung hierarchisch geschachtelter Terme erlauben. Terme können ihrerseits Kontext- und Event-Parameter mit Literalen oder untereinander verknüpfen, sodass die Navigation nach Belieben an Zustände von Komponenten, der Anwendung oder den Kontext gebunden werden kann. Neben binären Vergleichsoperatoren, wie $>, <, =, \leq, \geq, \neq$, werden auch Mengenoperatoren, wie `CONTAINS`, und Operatoren zur Auswertung von Typbeziehungen (`TYPE`, `ISA`) unterstützt.

Das View-Konzept adressiert die Notwendigkeit strukturierter Oberflächen und erlaubt die Modellierung mehrstufiger Dialoge zur Abbildung typischer Workflows.

Im Gegensatz zu bestehenden Web-Engineering-Modellen (vgl. Abschnitt 3.2), die sich i. d. R. auf Nutzerinteraktionen als Auslöser beschränken, können hiermit erstmals Sichtwechsel als Reaktion auf Zustandsänderungen und Systemereignisse definiert werden. Auf Basis der vorgestellten Modellklassen kann die Navigationsstruktur als endlicher, gerichteter Graph aufgebaut und durch Modellprüfer validiert werden. So können beispielsweise die Erreichbarkeit von Views getestet oder „Sackgassen" identifiziert werden, die zwar in Views hinein, aber nie wieder heraus führen. Dies kann, je nach Anwendungsfall, natürlich erwünscht sein, z. B. bei mehrstufigen Formularen, die keine Rücknavigation erlauben. Deshalb obliegt die Inspektion und letzte Entscheidung über die Korrektheit dem Autor der Komposition selbst.

5.3 Modellierung von adaptivem Verhalten

Ein weiteres Ziel der Arbeit besteht in der Unterstützung von kontextsensitivem Verhalten der modellierten Anwendungen. Wie in Kapitel 3 deutlich wurde, ist die direkte Anwendbarkeit etablierter Methoden und Techniken aus dem Bereich der *Adaptive Hypermedia* jedoch nicht gegeben, da in kompositen Anwendungen keine Annahmen über (vor-annotierte) Inhalte und genutzte Technologien getroffen werden können. Deshalb ist es nötig, die Adaptionsmethoden und Techniken zu aktualisieren und auf die Rahmenbedingungen von Mashups abzubilden. Davon sind weniger die abstrakten Methoden betroffen, als vielmehr die spezifischen Techniken, die nur mehr die öffentlichen Komponentenschnittstellen sowie die im Kompositionsmodell repräsentierte Verknüpfungslogik adressieren können.
Der nächste Abschnitt widmet sich der Vorstellung und Kategorisierung entsprechender Adaptionstechniken. Danach wird das *Adaptivity Model* vorgestellt, welches die kontextabhängige Konfiguration von Komponenten sowie die aspektorientierte Modellierung adaptiven Verhaltens von Kompositionen ermöglicht.

5.3.1 Adaptionstechniken für komposite Webanwendungen

Aus dem Anwendungsszenario der adaptiven Reiseplanung in Abschnitt 2.2.3 wurde deutlich, dass sich Adaptionstechniken auf einzelne Komponenten, auf mehrere Komponenten gleichzeitig (Nutzung einer einheitlichen Sprache) oder auf Kompositionsaspekte (Layout) beziehen können. Adaptionstechniken werden deshalb im Rahmen dieser Arbeit als jegliche Form der kontextabhängigen Änderung einer Anwendungskomposition bzw. eines Anwendungsmodells betrachtet.
Adaption stellt einen zusätzlichen Aspekt dar, der auf ein bestehendes Kompositionsmodell angewendet werden kann. Die Gründe hierfür können adaptiver, korrektiver, perfektiver oder auch extensiver Natur [Ketfi et al., 2002] sein, doch nicht alle diese Gründe können durch die Modellierung bedacht werden. So adressiert die extensive Adaption definitionsgemäß Zustände, die zur Modellierungszeit nicht vorhersehbar sind. Auch die korrektive Adaption, also die Sicherstellung der vollständig funktionalen Lauffähigkeit, kann i. d. R. nur im Rahmen der Fehlerbehandlung zur Laufzeit erfolgen. Die explizite Modellierung von adaptiver und perfektiver Adaption ist hingegen sinnvoll und wünschenswert. Sowohl die Reaktion auf wechselnde (Kontext-)Gegebenheiten (adaptiv) als auch die Angabe von zu optimierenden Kenngrößen, wie z. B. Antwortzeiten (perfektiv) können bei der Modellierung bedacht werden.

In ihrer Gesamtheit umfassen die möglichen Anpassungen zur Laufzeit die folgenden Techniken, die sowohl bei der Modellierung als auch durch die Laufzeitumgebungen adäquat unterstützt werden müssen.

5.3.1.1 Adaption auf Komponentenebene

Hinzufügen von Komponenten unterstützt die perfektive und extensive Adaption. Neue Komponenten können zusätzliche Daten enthalten, um den Nutzer bei seiner Aufgabe zu unterstützen. Es müssen Mechanismen existieren, um die dynamische Kopplung mit der restlichen Anwendung zu ermöglichen.

Entfernen von Komponenten kann die perfektive Adaption unterstützten, falls Komponenten nicht mehr benötigt werden. Das Entfernen fehlerhafter Komponenten kann auch korrektiven Aspekten dienen, stellt aber dann die Funktionsfähigkeit der Anwendung in Frage. In jedem Fall muss sichergestellt werden, dass das Entfernen nicht die Stabilität und Ausführung der Gesamtanwendung beeinträchtigt, und dass es nicht zum Datenverlust kommt.

Austausch von Komponenten kombiniert die beiden genannten Techniken zur korrektiven und adaptiven Adaption. Der Austausch birgt zwei wesentliche Herausforderungen: 1) den Zustandstransfer von der alten zur neuen Komponente, und 2) die Verhinderung von Nebenläufigkeit. Die gleichzeitige Ausführung der alten und neuen Komponente kann zu Problemen, wie Datenverlust, führen.

Rekonfiguration von Komponenten wirkt sich auf die Belegung der Eigenschaften einer Komponente aus. Ihre Veränderung impliziert einen Zustandswechsel der Komponente, der verschiedenartige Auswirkungen haben kann, beispielsweise die Änderung verwendeter Algorithmen, Datenquellen, visueller Gestaltungsmittel oder Modalitäten. Da Komponenten „Black Boxes“ darstellen, ist diese Form der Adaption auf Szenarien beschränkt, die sich auf Properties abbilden lassen bzw. für die Komponentenentwickler entsprechende Properties bereitstellen. Aus Sicht der Komposition stellt die Rekonfiguration von Komponenten die einfachste Möglichkeit zur Adaption dar. Da aber jede Komponente für sich ein u. U. adaptives Softwaresystem repräsentiert, kann die interne Interpretation der Property-Änderung, dennoch vergleichsweise mächtig ausfallen.

Mediation von Komponenten-Schnittstellen beschreibt die Anpassung der äußeren Schnittstelle einer Komponente an die Gegebenheiten des Kompositionsmodells. Dies wird nötig, wenn Komponenten an die Stelle von Templates (vgl. Abschnitt 5.2.1.1) treten, die zwar semantisch aber nicht syntaktisch äquivalent sind. Diese Technik umfasst vor dem Hintergrund des vorgestellten Komponentenmodells die Anpassung von Properties, Events und Operations mit Hilfe entsprechender Adapter oder Mediationsmechanismen, wie sie z. B. mit ParameterMappings (Abschnitt 5.2.2) vorgestellt wurden.

5.3.1.2 Adaption auf Kompositionsebene

Anpassung der Kommunikationbeziehungen zwischen Komponenten ist eine mächtige Technik, die es erlaubt, den Daten- und Kontrollfluss einer Anwendung kontextadaptiv zu gestalten. Komponenten können in bestehende Kommunikationsbeziehungen bzw. Links „eingehängt“ oder davon abgekoppelt werden.

Adaptives Layout bezieht sich auf die UI-Komponenten einer Anwendung, deren Anordnung auf der Oberfläche kontextabhängig definiert wird. Dies schließt nicht nur die Positionierung, sondern auch die Veränderung der Größe von Komponenten im Layout ein.

Anpassung der Navigation ist im Kontext von RIA differenzierter zu betrachten. In vornehmlich dokumentenzentrierten Ansätzen werden in erster Linie Angebot und Auszeichnung von Hyperlinks adaptiert. In komponentenbasierten Anwendungen herrscht allerdings keine Unterteilung in verschiedene Seiten mehr vor. Vielmehr bilden die verschiedenen Sichten (Views, Abschnitt 5.2.4) den Ansatzpunkt für Adaptionen. Diese können hinsichtlich der auslösenden Events, der referenzierten Layouts oder ihrer Bedingungen angepasst werden. Bedingungen können ihrerseits kontextadaptiv gestaltet sein, indem sie an Kontextparameter gebunden werden (z. B. „*Löse eine Transition bei Event E aus, falls der Nutzer laut Kontextmodell mehr als x Minuten angemeldet ist.*“).

Darüber hinaus kann sich die Adaption auf die Anwendungs- und Systemarchitektur auswirken. Dazu zählt die dynamische Migration von Komponenten oder Modulen zwischen Client und Server oder verschiedenen Servern im Sinne des Lastausgleichs. Wie bereits zu Beginn der Arbeit in Abschnitt 1.2 deutlich gemacht wurde, stehen die letztgenannten Techniken allerdings nicht im Fokus dieser Arbeit.
Die im Folgenden vorgestellten Modellierungsmittel dienen der Repräsentation von Adaptionslogik auf der Komponenten- und Kompositionsebene. Wie im Rahmen der Validierung gezeigt wird, können damit alle Anpassungen des in Abschnitt 2.2.3 skizzierten Szenarios vollständig abgedeckt werden.

5.3.2 *Adaptivity Model* – Aspektorientierte Modellierung anwendungsbezogener Adaptivität

Das *Adaptivity Model* illustriert die Erweiterbarkeit des vorgestellten Metamodells, indem es den Aspekt der Kontextabhängigkeit durch die Modellierung adaptiver Verhaltenslogik in die Komposition einbringt. Abbildung 5.20 zeigt das optionale Teilmodell im Überblick. Es dient der Repräsentation des zur Anpassung relevanten Kontextes (*Context*), der direkten Verknüpfung von Kontextdaten und Komponenten (*Context Links*) sowie der aspektorientierten Definition von Anpassungen der Komposition in Abhängigkeit von Kontextänderungen (*Adaptation Aspects*).

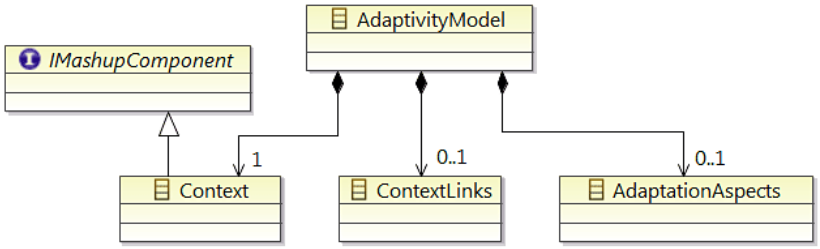

Abb. 5.20: Das *Adaptivity Model* im Überblick

Im Folgenden werden die genannten Bestandteile näher erläutert.

5.3.2.1 Repräsentation des Kontextmodells und adaptive (Re-)Konfiguration

Ein wesentlicher Bestandteil des Adaptivity Models ist die Klasse *Context*, die die Schnittstelle zum Kontextmodell darstellt. Letzteres wird als Komponente repräsentiert, d. h. es verfügt über Properties, die Kontexteigenschaften entsprechen, welche mittels Operationen geändert werden können und deren Änderungen in Form von Events publiziert werden. Dieses einfache Prinzip bildet die Grundlage für die im nächsten Abschnitt vorgestellten Adaptivitätsmechanismen sowie für die direkte Kontextualisierung von Komponenten.

Die Kontextkomponente enthält die Gesamtheit aller zur Kontextualisierung referenzierter Kontextparameter. Bei der Modellierung sind verschiedene Herangehensweisen denkbar. Einerseits kann der Kontextdienst über eine (S)MCDL-Beschreibung repräsentiert werden, was die einfache Integration in das Kompositionsmodell erlaubt. Andererseits ist es ebenso möglich, die Context-Klasse nur mit den Properties zu instanziieren, die tatsächlich benötigt werden.

Die Repräsentation des Kontextes als Mashup-Komponente nach dem vorgestellten Modell bietet eine Reihe von Vorteilen. Zunächst vereinfacht sie die Referenzierung von Kontexteigenschaften, da auf konkrete Properties im Modell zugegriffen werden kann, und keine abstrakten Zugriffspfade formuliert werden müssen, wie bei gängigen Adaptive-Hypermedia-Konzepten. Zudem impliziert sie, dass Parameter durch entsprechende Domänenmodelle eindeutig semantisch typisiert sind und eine Abbildung auf einen syntaktischen Datentyp (Grounding) existiert. Somit können die gleichen Mechanismen zur Steigerung der Interoperabilität zum Einsatz kommen, wie bei der Kopplung der sonstigen Mashup-Komponenten, d. h. syntaktische Inkompatibilitäten zwischen Kontext- und Komponenteneigenschaften können über die semantische Ebene überbrückt werden (vgl. Abschnitt 6.2.4.2).

Die Nutzung von Kontextwissen kann im Adaptivity Model auf zweierlei Arten erfolgen: **(1)** Zum einen kann auf Kontextänderungen mit einer der im letzten Abschnitt vorgestellten Adaptionstechniken reagiert werden, die u. U. zur Beeinflussung des Kompositionsmodells führen. Die Formulierung derartiger Adaptionslogik erfolgt aspektorientiert und wird im nächsten Abschnitt erläutert. **(2)** Zum anderen können Komponenten über sog. *Context Links* direkt mit Kontexteigenschaften verknüpft bzw. kontextualisiert werden.

Das Konzept der **Context Links** ermöglicht die kontextabhängige Konfiguration von Komponenten, indem ihre Eigenschaften direkt mit Context-Properties verbunden werden. Die Verknüpfung erfolgt ähnlich der Modellierung des Datenflusses im Communication Model (Abschnitt 5.2.2.1) und setzt die Übereinstimmung der verknüpften Datentypen voraus. Context Links stellen unidirektionale 1:n-Verknüpfungen zwischen einem Kontextdatum und potentiell mehreren Komponenteneigenschaften dar. Wie Abbildung 5.21 zeigt, erfolgt keine bidirektionale Synchronisation wie bei Property Links – vielmehr führt nur die Änderung einer Kontexteigenschaft zur Verbreitung an alle angeschlossen Parteien. Der Sinn dieser Einschränkung erschließt sich am Beispiel: Die initiale Bindung der Sprache einer Komponente an die Muttersprache des Nutzers (repräsentiert im Kontextmodell) ist häufig sinnvoll – die Änderung der Sprache durch den Nutzer (in einer Komponente) bedeutet jedoch i. d. R. nicht, dass sich damit seine Muttersprache im Kontextmodell ändern sollte.

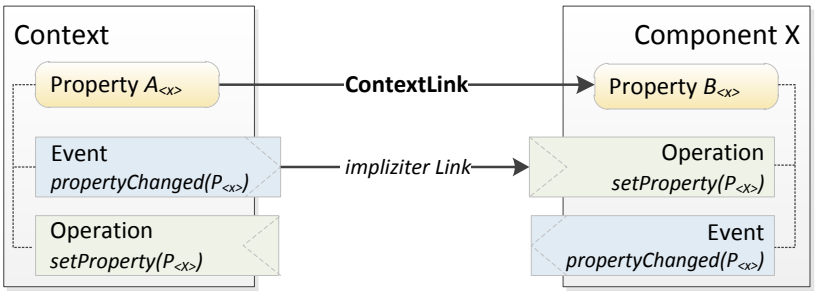

Abb. 5.21: Kontextualisierung von Komponenten über *Context Links*

Der Kompositeur hat Kontrolle über die Aktualisierungshäufigkeit bei Kontextänderungen. Zum einen kann er über das Attribut `isInitial` eines Links bestimmen, ob die Belegung der angeschlossenen Eigenschaften nur initial, oder fortwährend mit dem Kontextmodell abgeglichen werden sollen. Die Zentrierung einer Karte auf die aktuelle Position mag beim Laden der Anwendung sinnvoll sein – sobald der Nutzer in der Karte navigiert, ist eine permanente Änderung zurück eher hinderlich ($isInitial = true$). In anderen Situationen ist dies aber ggf. wünschenswert, z. B. im Fall einer adaptiven Touristeninformation, die Information zu Sehenswürdigkeiten in der näheren Umgebung liefert. Hier sollten die Komponenten möglichst permanent über den Standort des Nutzers informiert werden ($isInitial = false$).
Um den damit einhergehenden Datenverkehr sinnvoll einzuschränken, können Context Links mit dem Attribut `threshold` versehen werden. Dieses gibt die minimale Zeitspanne zwischen zwei Aktualisierungen (in Sekunden) an. Beim o. g. Beispiel der Touristeninformation ist in Anbetracht der Geschwindigkeit der Touristen beispielsweise keine sekündliche Anpassung nötig. Die manuelle Angabe des Intervalls ist optional und nicht in jedem Fall nötig - Context Properties können zudem „von Haus aus“ über einen Standardwert verfügen, der im Modell durch das Attribut `defaultThreshold` ausgedrückt wird.

Context Links können letztlich als Vereinfachung gegenüber den im nächsten Abschnitt vorgestellten Adaptionsaspekten verstanden werden. Die direkte Verbindung von Kontexteigenschaften und Kontextparametern entlastet Entwickler von der Formulierung entsprechender Rekonfigurationsaspekte und hält das Modell einfach und verständlich. Zudem kann die Erstellung der Links durch Autorenwerkzeuge unterstützt werden, indem ein Typvergleich zwischen Komponenten- und Kontexteigenschaften erfolgt. Die Möglichkeiten zur Beschränkung der Aktualisierungshäufigkeit kommen der Performanz und Usability zugute: Die Anzahl der ausgetauschten, zu prüfenden und ggf. zu transformierenden Daten wird minimiert, und eine ständige Rekonfiguration zu Lasten der Benutzbarkeit vermieden.

5.3.2.2 Aspektorientierte Modellierung von Adaptionslogik

Nach dem Vorbild von UWE (Abschnitt 3.2.1) erfolgt die Modellierung des adaptiven Verhaltens aspektorientiert. Die explizite Auszeichnung von Komponenten und anderen Konzepten des Modells als adaptiv würde dem Anliegen der SoC widersprechen und zur Vermischung von Anwendungs- und Adaptionslogik führen. Das Prinzip der aspektorientierten Adaption ermöglicht es hingegen, Veränderungen des Modells –

und somit implizit der Anwendung – ohne Beeinflussung der bislang vorgestellten Teilmodelle zu definieren und darauf anzuwenden. Die Entkopplung erlaubt die unabhängige Definition von Adaptionsaspekten, fördert die Les- und Wartbarkeit und steigert die Wiederverwendbarkeit von Adaptionslogik, da die Aspekte in verschiedene Kompositionsmodelle eingebunden werden können.

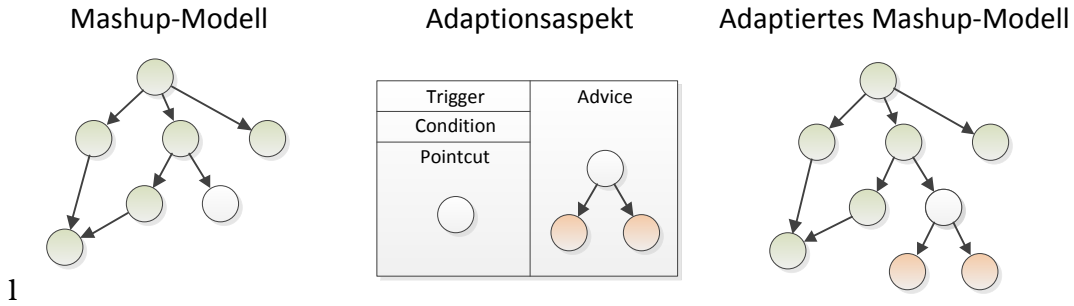

Abb. 5.22: Prinzip der aspektorientierten Beschreibung von Adaption

In Abbildung 5.22 ist das weiße Modellelement (links) durch den *Pointcut* des Aspekts (mitte) als Ziel der Adaption adressiert. Die Ausführung eines Aspekts wird durch einen *Trigger* ausgelöst und kann an eine Bedingung (*Condition*) gebunden sein. Nach der Anwendung des Aspekts enthält das Modell an der spezifizierten Stelle die durch den *Advice* definierten Änderungen (rechts). Diese können neue Modellelemente umfassen, aber auch bestehende ändern oder entfernen.

Abbildung 5.23 zeigt die wichtigsten Klassen des Adaptivity Models und ihre Zusammenhänge. Kernkonzept bildet der Adaptionsaspekt, der durch die Klasse AdaptationAspect beschrieben wird. Er repräsentiert eine Modelländerung der kompositen Anwendung zur Laufzeit. Als Auslöser (Trigger) dieser Anpassung werden Events des Modells referenziert. Analog zum Wechsel zwischen Sichten im Screenflow-Modell (vgl. Abschnitt 5.2.4) kann die Adaption also gleichsam durch Nutzerinteraktionen, Kontextänderungen oder die Laufzeitumgebung ausgelöst werden, da all diese Bestandteile dem gleichen Komponentenmodell unterliegen. Jede Adaption kann Bedingungen unterliegen, deren Beschreibung in Form der Klasse Condition auch im Screenflow Model zum Einsatz kommt. Werden alle Bedingungen erfüllt, so erfolgt die Anpassung des im Pointcut adressierten Modellelements gemäß der im Advice definierten Technik. Die Reihenfolge gleichzeitig ausgelöster Aspekte kann durch die Angabe von (numerischen) Prioritäten beeinflusst werden.

Die folgenden Abschnitte beleuchten die aspektorientierte Modellierung und die in der Abbildung dargestellten Modellkonstrukte näher.

Trigger – Auslöser der Adaption

Über die Klasse Trigger wird der Auslöser eines Adaptionsaspektes festgelegt. Dabei handelt es sich um Verweise auf ein Event oder ein Property. Die Referenz auf letztere impliziert, dass das entsprechende *Change*-Event Auslöser der Adaption sein soll. Existieren mehrere Trigger, so sind sie implizit `ODER`-verknüpft, d. h. jeder von Ihnen hat die Ausführung des Aspektes zur Folge.

Die einheitliche Nutzung des Komponentenmodells für die verschiedenen Bestandteile einer Komposition wirkt sich auch hier vorteilhaft aus. Der Auslöser kann

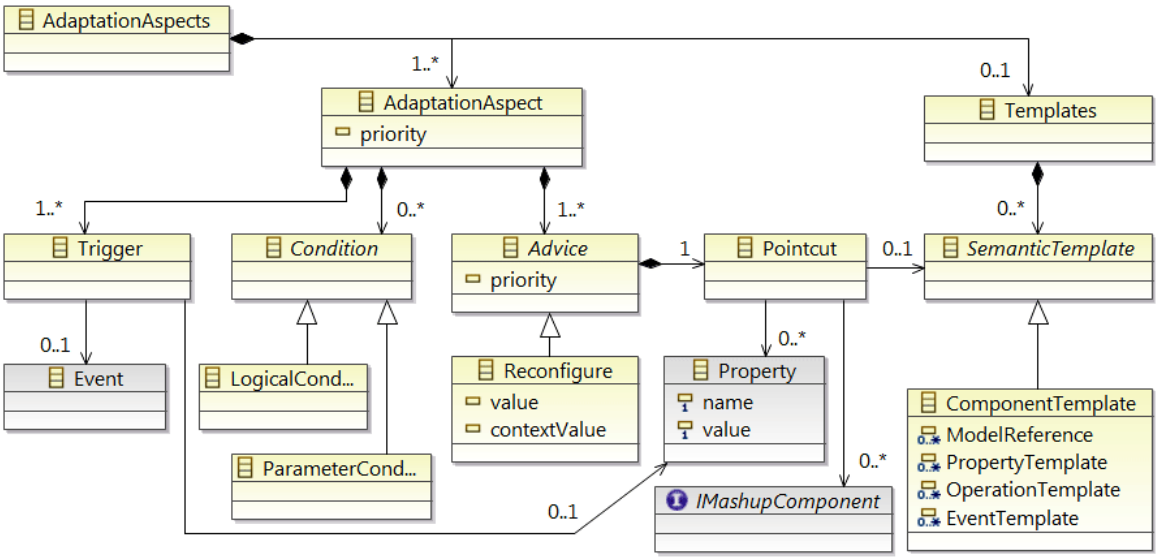

Abb. 5.23: Modellierung adaptiven Verhaltens durch Adaptionsaspekte

Teil der Schnittstelle der Laufzeitumgebung oder Rahmenanwendung (Runtime, Abschnitt 5.2.1) sein. Gleichermaßen können Ereignisse/Eigenschaften von integrierten Komponenten der Anwendung adressiert werden. Schließlich bildet die Nutzung von Kontexteigenschaften den häufigsten Anwendungsfall.

Mit Bezug auf das in Abschnitt 2.2.3 letztgenannte Beispiel würde der Trigger einen Verweis auf die Eigenschaft *BatteryLevel* der Kontextkomponente umfassen.

Auch die Angabe komplexer Trigger, z. B. die Formulierung von Sequenzen oder sonstigen kausalen Abhängigkeiten zwischen Ereignissen als Auslöser, ist möglich und im Modell als Erweiterungspunkt vorgesehen. Die Angabe entsprechender Logik soll jedoch nicht im Mittelpunkt der weiteren Betrachtung stehen.

Conditions – Einschränkung von Adaptionen

Mit Adaptionsaspekten können Beschränkungen verbunden sein, die durch die Klasse Condition ausgedrückt werden. Diese kam bereits im Zusammenhang mit dem Screenflow-Modell (ViewTransitions, Abschnitt 5.2.4) zur Sprache. Die gleichen Modellkonstrukte zur Überprüfung von Event-Parametern (ParameterCondition) – hier bezogen auf den Trigger – und zur booleschen Verknüpfung hierarchisch geschachtelter Terme (LogicalCondition) können auch hier genutzt werden. So kann beispielsweise ausgedrückt werden, dass ein Aspekt zwar durch die Änderung eines bestimmten Kontextparameters ausgelöst wird, die Anpassung jedoch nur über einem in der Condition definierten Grenzwert tatsächlich ausgeführt wird.

Am oben aufgegriffenen Beispiel bedeutet dies, dass nach Auslösen des Aspektes durch das Event *BatteryLevelChanged* nun geprüft werden kann, ob der gelieferte Ladezustand unter 20% liegt. Zusätzlich kann sichergestellt werden, dass das genutzte Gerät *mobil* ist, indem die entsprechende Kontexteigenschaft mit einem Literal verglichen wird. Bei der Angabe mehrerer Conditions sind diese `AND`-verknüpft, d. h. alle Bedingungen müssten erfüllt sein, bevor der Aspekt angewendet wird.

Pointcuts – Adressierung von Adaptionsobjekten

Sind alle Bedingungen eines Aspektes erfüllt, kann die Ausführung der zugehörigen *Advices* beginnen. Diese beziehen sich jeweils auf einen Pointcut, der wie in der traditionellen aspektorientierten Programmierung der Adressierung dient. Hier enthält er den Verweis auf die Bestandteile des Kompositionsmodells, die durch den Advice verändert werden sollen.

In Abschnitt 5.3.1 wurden verschiedene Ziele von Adaptionstechniken identifiziert, u. a. Properties, Komponenten, Links, Layouts, usw. All diese Entitäten können direkt durch den Pointcut referenziert werden, wie in Abbildung 5.23 exemplarisch für die Klassen Property und IMashupComponent gezeigt wird. Im vollständigen Modell besteht dazu ein Verweis auf das gemeinsame Interface IAdaptable, welches durch alle potentiellen Ziele einer Adaption implementiert wird.

Pointcuts können allerdings auch abstrakter Natur sein, indem sie nicht auf konkrete Bestandteile, sondern auf semantische Vorlagen verweisen, die unter Templates definiert werden. Jeder Modellbestandteil, der einer solchen Vorlage entspricht, wird dann wie im Advice beschrieben angepasst. Durch die Abstraktion kann Redundanz in den Aspekten vermieden werden. Vorlagen können aspektübergreifend genutzt werden und machen die direkte Adressierung mehrerer, gleichartiger Modellbestandteile unnötig. Insbesondere in Kompositionen, die sich zur Laufzeit ändern, kann somit adaptives Verhalten sichergestellt werden, ohne dass die integrierten Komponenten zur Entwicklungszeit bekannt sein müssen.

Abbildung 5.23 zeigt am Beispiel von ComponentTemplate, wie die Definition einer Komponentenvorlage erfolgt. Es wird klar, dass Vorlagen hierarchisch geschachtelt sein können. Eine Komponente kann durch semantische Referenzen der Komponentenfunktionalität (ModelReference) sowie durch Properties, Operations und Events beschrieben sein, für die jeweils entsprechende Templates definiert werden.

Im o. g. Beispiel (niedriger Batteriestand) kann ein direkter Pointcut auf die Komponente „VideoPlayer“ verweisen, die ausgetauscht werden soll. Falls ein ComponentTemplate mit der gleichen Schnittstelle genutzt wird, welches über die funktionale Modellreferenz zur Videoausgabe ausgezeichnet ist, wird der Aspekt auf alle Video-Komponenten der Anwendung angewendet.

Advices – Definition von Adaptionsaktionen

Advices repräsentieren schließlich die auszuführenden Adaptionsaktionen im Kompositionsmodell, die 1:1 mit den in Abschnitt 5.3.1 vorgestellten Techniken korrespondieren. Eine Ausnahme bildet die Mediation, die durch die Laufzeitumgebung erfolgt, da eventuell zu vermittelnde Schnittstellen vor der Auswahl der Komponenten zur Laufzeit nicht bekannt sind. Jede Technik wird durch eine spezifische Implementierung von Advice umgesetzt, die über entsprechende Attribute verfügt. In Abbildung 5.23 ist exemplarisch die Aktion Reconfigure zu sehen, die den im Pointcut adressierten Komponenteneigenschaften einen neuen fest Wert (`value`) oder Kontextwert (`contextValue`) zuweist.

Dabei wird klar, dass das adressierte Modellelement im Pointcut und die Adaptionstechnik im Advice in Beziehung stehen. Die Korrektheit dieser Zusammenhänge muss zusätzlich im Modell, z. B. in Form von OCL-Constraints, sichergestellt werden. Die Ausführungsreihenfolge der Advices eines Aspekts kann wiederum durch die Angabe von Prioritäten beeinflusst werden.

Adaptionstechniken – und somit Advices – können unterschiedliche Modellbestandteile adressieren. Auf eine umfassende Darstellung aller existierenden Klassen wird an dieser Stelle mit Verweis auf die Zusammenfassung der Techniken in Abschnitt 5.3.1 verzichtet. Einige Beispiele aus dem in Abschnitt 2.2.3 geschilderten adaptiven Szenario sollen jedoch einen praktischen Einblick geben:
Der geforderte Austausch einer Video-Komponente durch eine Bildergalerie in Abhängigkeit vom Batteriestatus wird durch die Klasse ExchangeComponent repräsentiert. Diese ersetzt die im Pointcut adressierte gegen eine neue Komponente, die a) über eine ID konkret definiert ist, b) der gleichen semantischen Schnittstelle genügt, oder c) einer Vorlage aus Templates entspricht. Im konkreten Fall wäre letztere Option zu wählen, da sich die Funktionalität von Ausgangs- und Zielkomponente unterscheiden und das Template aufgrund der dynamischen Discovery eine größere Flexibilität als die konkrete Referenzierung einer Alternative verspricht.
Für den Fall, dass die Wetterkomponente auf einem kleinen Bildschirm lediglich ausgeblendet werden soll, kann die Sichtbarkeit durch den Advice SetVisibility und dessen Attribut `visibility` beeinflusst werden. Die vollständige Entfernung aus der Komposition wird hingegen durch die Klasse RemoveComponent erreicht.
Auch Veränderungen des Layouts lassen sich über Advices formulieren. Durch die Klasse ReconfigureLayout können im Pointcut adressierte Layouts rekonfiguriert und ausgetauscht werden. Gleichsam kann das Master-Layout eines Views durch die Aktion ReconfigureScreenflow umgeschaltet werden. Hierbei kann auf eine Vereinfachung zurückgegriffen werden: Neben der Angabe der Alternativen im Advice kann auf ContextualScreenflows verwiesen werden, die kontextabhängige Sichten auf die Anwendung darstellen. Wie Abbildung 5.18 zeigt, können sie dem Screenflow Model hinzugefügt werden. Gegenüber „normalen“ Screenflows unterscheiden sie sich durch die Angabe einer Bedingung, unter der sie geladen werden sollen. Die Entscheidung erfolgt beim Anwendungsstart, da sie i. d. R. abhängig von Endgeräteklassen und Nutzern definiert wird, die sich zur Laufzeit nicht ändern. Ist der Wechsel von Screenflows zur Laufzeit gewünscht, kann dies dennoch durch Aspekte modelliert werden.

Abbildung 5.24 zeigt das Zusammenspiel der vorgestellten Klassen schematisch am Referenzbeispiel: Der Aspekt „LowBattery-NoVideo“ wird ausgelöst, sobald sich der Ladezustand des Endgerätes des aktuellen Nutzers (*BatteryLevel*) verändert. Daraufhin wird die logische Bedingung ausgewertet, die überprüft, ob der Ladezustand unter 20 % liegt und ob es sich um ein mobiles Gerät handelt. Ist beides der Fall, wir der Advice *ExchangeComponent* ausgeführt. Die durch den Pointcut adressierte Komponente „VideoPlayer“ wird daraufhin durch eine alternative Implementierung ersetzt. Da das Attribut `currentTemplate` auf *false* gesetzt ist, wird nicht der VideoPlayer als Template verstanden, sondern auf ein separat definiertes *ComponentTemplate* verwiesen. Im Beispiel unterscheidet sich lediglich die geforderte Funktionalität – sollten adressierte Komponente und Template unterschiedliche Schnittstelle besitzen, muss durch weitere Advices ggf. die Rekonfiguration angeschlossener Links etc. erfolgen.
Der Erkennung und Behandlung von Konflikten zwischen Adaptionsaspekten widmen sich verschiedene aktuelle Forschungsansätze, u. a. von Karol et al. (2011). Derartige Mechanismen bilden nicht den Schwerpunkt der vorliegenden Arbeit, können aber aufgrund der konzeptionellen Nähe zukünftig integriert werden. Die

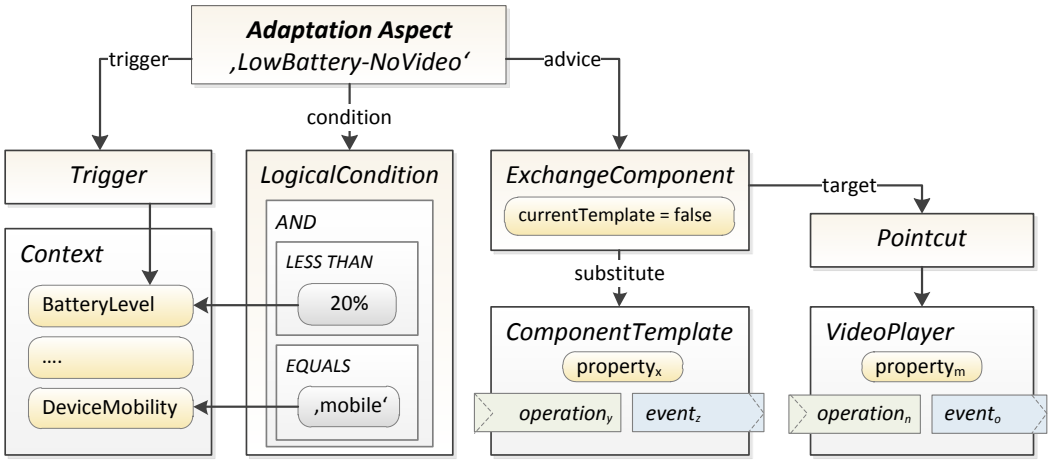

Abb. 5.24: Beispiel eines Adaptionaspektes zum Komponententausch (schematisch)

wichtigsten Anforderungen zur Sicherstellung der Korrektheit von Adaptionen können mit Hilfe des Modells bereits jetzt adressiert werden:
Zur Sicherung der *Konfluenz*, d. h. der Unabhängigkeit der resultierenden Adaption von der Ausführungsreihenfolge der Aspekte oder Advices, muss zur Entwicklungszeit die Analyse von Pointcuts und Aktionen auf Schnittmengen erfolgen. Durch die Anwendung von Prioritäten auf Aspekte und Advices kann im Konfliktfall die Sicherstellung der korrekten Reihenfolge erfolgen. *Konsistenz* ist dann gewährleistet, wenn Aspekte widerspruchsfrei sind. Die Konsistenz von Pointcuts und Advices – insbesondere, ob sich ein Advice und Pointcut auf die gleiche Modellklasse beziehen – kann automatisch getestet werden. Durch eine Grenzwertprüfung können zudem bestimmte Widersprüche in Bedingungen ausgeschlossen werden. Dies hat jedoch Grenzen, sodass komplexe Bedingungen einer gesonderten Fehlerbehandlung unterzogen werden müssen. Durch die *Terminierung* wird schließlich gesichert, dass Aspekte in endlicher Zeit zu einem Adaptionsergebnis zu führen. Problematisch kann hierbei die Änderung von Kontexteigenschaften sein, da diese weitere Events und somit Adaptionen nach sich zieht, wodurch es zu Endlosschleifen kommen kann. Durch die Auswertung des MCDL-Attributs `trigger` können derartige Zirkelbezüge teilweise erkannt werden – vollständig ausgeschlossen sind Konflikte trotz allem nicht, da das Wissen über interne Zusammenhänge der Komponenten fehlt. Eine Behandlung durch die Laufzeitumgebung ist deshalb nötig.

5.4 Ablauf und Unterstützung bei der Modellierung

In den vorherigen Teilkapiteln wurden Konzepte vorgestellt, die *UI Mashup Composern* (vgl. Abbildung 4.1) die Spezifikation aller zur Ausführung notwendigen Anwendungsbelange auf einer abstrakten, plattformunabhängigen Ebene ermöglichen. Auch wenn der Entwurfsprozess nicht im Vordergrund dieser Arbeit steht, soll der folgende Abschnitt die Einbindung der Modellkonzepte in den modellgetriebenen Entwicklungsprozess und dessen prinzipiellen Ablauf bis zur Bereitstellung einer Mashup-Anwendung verdeutlichen.
Bei der Mashup-Erstellung auf der Daten-Ebene können drei Schritte unterschieden werden. (1) Zunächst werden zu nutzende Quellen bzw. Ressourcen identifiziert und

ausgewählt, (2) danach werden sie miteinander verknüpft (3) und schließlich anwendungsspezifisch konfiguriert. Die universelle Modellierung nach dem o. g. Modell erfolgt auf ähnliche Weise durch die Auswahl von Komponenten, ihre Verknüpfung und Konfiguration. Zusätzlich muss die Modellierung der Präsentationsaspekte erfolgen, wie die Definition von Stilen, Layouts und Sichten.

Aus Sicht der modellgetriebenen Entwicklung (Abschnitt 2.1.3) entspricht das vorgestellte Kompositionsmodell dem Platform Independent Model (PIM) einer Anwendung, da zur Erstellung keine plattformabhängigen Entwurfsentscheidungen getroffen werden müssen. Abbildung 5.25 zeigt die prinzipiellen Abhängigkeiten zwischen den Teilmodellen, die implizit den Erstellungsprozess vorgeben. Die Pfeile zeigen Abhängigkeiten und Nutzungsbeziehungen zwischen diesen an. Es ist u. a. ersichtlich, dass fast immer auf Konzepte aus dem Conceptual Model Bezug genommen wird und das Adaptivity Model auf Modellklassen aller Teilmodelle verweisen kann. Drei Teilmodelle sind mit einem Zahnrad versehen – ihre Erstellung kann durch Generierungslogik unterstützt werden.

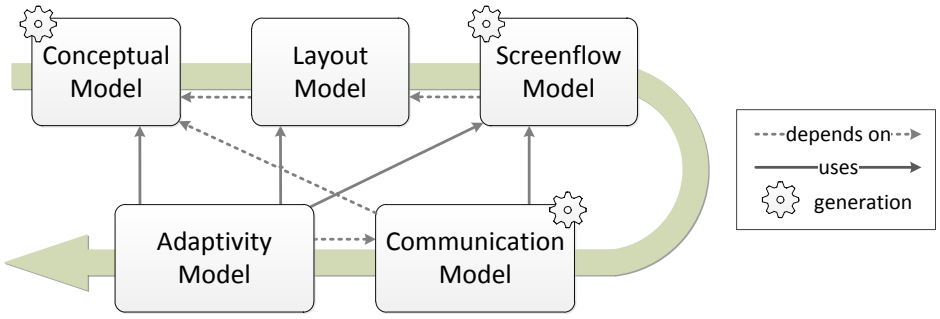

Abb. 5.25: Ablauf und Abhängigkeiten im Modellierungsprozess

Die Abhängigkeiten und Unterstützungsmöglichkeiten illustriert die folgende Vorstellung der Entwicklungsschritte:

Auswahl und Konfiguration von Komponenten Zunächst muss das Conceptual Model definiert werden, welches Grundlage für alle weiteren Modelle bietet. Hauptaufgabe dabei ist die Identifikation und Auswahl der Komponenten, z. B. mit Hilfe von Such- und Empfehlungsverfahren. Ihre Integration kann vollautomatisch erfolgen, d. h. die benötigten Informationen werden aus der MCDL-Beschreibung entnommen und auf Modellklassen abgebildet. Auch mit WSDL oder WADL beschriebene Dienste können direkt in das Modell übertragen werden – die automatische Kapselung als Komponenten hat jedoch, wie in Abschnitt 5.1.4 erwähnt, Grenzen, sodass ein funktionaler Test unablässig ist. Nach der Auswahl konkreter Komponenten kann die Abstraktion zu Templates erfolgen. Letztere können auch manuell erstellt werden, was jedoch den weniger praktikablen Weg darstellt. Im letzten Teilschritt erfolgt die Konfiguration der Komponenten durch die Belegung ihrer Properties mit konkreten Werten.

Definition des Layouts Im nächsten Schritt wird die Anordnung von Komponenten auf der Benutzeroberfläche definiert. Hier kann das Wissen aus dem Conceptual Model nur bedingt helfen. Die Modellierung muss durch den Entwickler erfolgen, der dazu auf die UI-Komponenten des Conceptual Models verweist.

Definition von Sichten Verschiedene Sichten bzw. Views können (semi-)automatisch aus dem Layout Model geschlossen werden. Hinter jedem modellierten Top-Level-Layout – d. h. es ist nicht Teil eines übergeordneten Layouts – steckt i. d. R. die Absicht, es dem Nutzer zu präsentieren. Es kann somit jeweils eine Sicht generiert werden, die auf das Layout verweist. Die Transitionen zwischen den Sichten muss schließlich der Entwickler selbst festlegen.

Verknüpfung von Komponenten Die vorgesehenen Komponenten oder Templates können nun durch Links verbunden werden. Die Grundstruktur des Communication Models kann wiederum semi-automatisch auf Basis der vorhandenen Operationen und Ereignisse erstellt werden: Für jede Parametersignatur kann ein getypter Link instanziiert werden, der mit den jeweils passenden Events und Operations verbunden wird. Falls Callback-Operations oder -Events definiert sind, wird entsprechend ein BackLink angelegt. Die Abbildung auf Links ist nicht immer eindeutig und muss deshalb bei Bedarf durch Entwickler angepasst oder erweitert werden, z. B. falls mehrere Links pro Signatur gewünscht sind oder Parameter-Mappings nötig werden. Links können schließlich mit einem Verweis auf die Views versehen werden, in denen sie aktiv sein sollen, um die Kommunikation bewusst auf bestimmte Sichten zu beschränken.

Spezifikation von Adaptionslogik Die Definition von Contextual Screenflows, Adaptionsaspekten, etc. muss schließlich durch den Entwickler erfolgen. Selbstverständlich ist auch hier umfangreiche Unterstützung durch das Autorenwerkzeug vorstellbar, z. B. indem die kontextadaptive Belegung von Komponenteneigenschaften vorgeschlagen wird, sobald eine semantische Typgleichheit mit einer Kontexteigenschaft festgestellt wird.

Das vollständige Kompositionsmodell kann durch Transformationen in plattformspezifische Modelle und schließlich und in Quellcode umgewandelt werden. Die direkte Interpretation des Modells durch die Plattformen ist gleichsam möglich und wird aufgrund fehlender einheitlicher Plattform-Modelle inzwischen auch von führenden Vertretern der MDA-Bewegung für den praktikabelsten Weg erklärt [Bezivin, 2011].

5.5 Zusammenfassung und Diskussion

Ausgehend von den in Abschnitt 2.3 aufgestellten Anforderungen wurden in diesem Kapitel Konzepte zur plattformunabhängigen Modellierung interaktiver Mashup-Anwendungen vorgestellt. Sie umfassen ein universelles Komponentenmodell zur Repräsentation von Web-Ressourcen und -Diensten als zustandsbehaftete Anwendungsbestandteile gemäß den Prinzipien komponentenorientierter Systeme. Darauf aufbauend wurde ein belangorientiertes Metamodell vorgestellt, das die plattformunabhängige Modellierung von Anwendungen aus diesen Komponenten ermöglicht. Schließlich wurde dem Ziel der Kontextadaptivität durch die Konzeption eines Teilmodells Rechnung getragen, welches die dynamische Anpassung von Kompositionen in Abhängigkeit von Kontextänderungen beschreibt.
Nachfolgend werden die Alleinstellungsmerkmale und die wissenschaftlichen Beiträge der beschriebenen Lösungen zusammengefasst und hinsichtlich der Anforderungen – jeweils mit ✪ markiert – bewertet.

Universelle Modellierung und Beschreibung von Mashup-Komponenten

Das in Teilkapitel 5.1 vorgestellte Komponentenmodell bildet die Grundlage für die Modellierung interaktiver, kompositer Webanwendungen. Es repräsentiert Web-Ressourcen und -Dienste – egal ob Teil der Präsentations- oder Geschäftslogikebene – als wiederverwendbare Komponenten anhand einheitlicher, technologieunabhängiger Abstraktionen und unterstützt somit die ✪ Plattformunabhängigkeit und ✪ Wiederverwendbarkeit bei der Entwicklung. Das Modell folgt dem *Information-Hiding-Prinzip* der ✪ Komponentenorientierung, d. h. Komponenten sind abgeschlossene Systeme, die modular, parametrisierbar und nur über klar definierte Schnittstellen ansprechbar sind. Ob sie Datenquellen, Geschäftslogik oder UI-Bestandteile darstellen, spielt aufgrund der ✪ Universalität des Modells keine Rolle. Sie können jedoch – im Gegensatz zur Mehrzahl existierender Dienst- und Mashup-Modelle – über einen Zustand verfügen, der durch Properties ausgedrückt und über Events publiziert wird. Das ereignisorientierte Paradigma bildet gleichzeitig die Basis für den Kontroll- und Datenfluss bzw. die ✪ Koordination innerhalb kompositer Anwendungen – auch über verschiedene Anwendungsebenen hinweg.

Als Formate zur ✪ Beschreibung von Komponenten wurden drei deklarative Sprachen in aufsteigender Mächtigkeit vorgestellt. Sie ermöglichen die Verwaltung und Suche von Komponenten zur Designzeit und stellen ebenso alle nötigen Informationen zur ✪ Discovery und Integration zur Laufzeit zur Verfügung. Bei der Beschreibung wird zwischen der abstrakten Schnittstelle und Bindungsinformationen für die jeweiligen Implementierungen unterschieden. Erstere wird zur ✪ Abstraktion genutzt und dient der ✪ Plattformunabhängigkeit. Letztere können Abbildungs- und Transformationsregeln enthalten, die syntaktische Differenzen zur öffentlichen Schnittstelle überbrücken und somit die ✪ Interoperabilität fördern.

Als XML-basierte Basisrepräsentation für Mashup-Komponenten wurde die MCDL vorgestellt, die sich hinsichtlich dem Aufbau und der Typisierung über XML-Schema an WSDL orientiert. Mit der SMCDL steht eine Erweiterung zur semantischen Typisierung und Annotation mit funktionalen und nicht-funktionalen Konzepten zur Verfügung. Die Nutzung von ✪ Semantik erfolgt in Anlehnung an die Konzepte der SWS (vgl. Abschnitt 3.1.2), wie SAWSDL und WSMO und erlaubt die Übertragung von Teillösungen hinsichtlich der kontextsensitiven ✪ Discovery und ✪ Mediation zur Laufzeit. Gegenüber konventionellen Diensten wird freilich ein erweitertes Vokabular genutzt, welches z. B. die Beschreibung von Integrations- und Interaktionskonzepten ermöglicht. Auch der Zustand von Komponenten stellt einen klaren Unterschied dar, der eigene Konzepte nötig macht. Die maximale Ausdrucksstärke bei der Beschreibung ist mit der ontologiebasierten Repräsentation MCDO gegeben, die u. a. die Formulierung von Vor- und Nachbedingungen erlaubt. Da sich alle Beschreibungsformate auf sie abbilden lassen, bildet sie später die Grundlage zur ✪ Modellverwaltung für Komponenten.

Belangorientierte Modellierung universeller Kompositionen

Zur Beschreibung interaktiver kompositer Webanwendungen wurde in Teilkapitel 5.2 ein Kompositionsmodell vorgestellt. Im Gegensatz zu konventionellen Web-Engineering-Ansätzen steht dabei die Komposition von funktionalen Bestandteilen entsprechend des o. g. Komponentenmodells, nicht die Kombination von Inhalten im Vordergrund. Zu diesem Zweck stehen Modellierungsmittel zur Verfügung, die neben

der Definition und Konfiguration aller enthaltenen Mashup-Komponenten die Spezifikation des Daten- und Kontrollflusses, der visuellen Erscheinung, der Navigation und Kontextadaptivität erlauben. Die ✪ Erweiterbarkeit auf Metamodellebene ist durch die konsequente Nutzung abstrakter Klassen und Interfaces gegeben. Bezüglich der Anforderungen nach ✪ Standardkonformität und Werkzeugunterstützung wird auf Abschnitt 7.1 zur Umsetzung des Modells verwiesen.

Basiskonzepte des Modells bilden die einzubindenden Mashup-Komponenten, die anwendungsspezifisch konfiguriert werden können. Die ✪ Erweiterbarkeit mit neuen Komponenten ist allein auf Modellebene und ohne jegliche Programmierung möglich. Darüber hinaus werden auch die Laufzeitumgebung und die Rahmenanwendung als Komponenten repräsentiert, was ihre Einbeziehung unter Nutzung einheitlicher Paradigmen ermöglicht (✪ Universalität). Auch bestehende Kompositionen können als Komponenten integriert werden, um die ✪ Wiederverwendbarkeit und Modellierung auf unterschiedlichen Abstraktionsstufen zu stärken.

Templates erlauben die semantische ✪ Abstraktion von Komponentenimplementierungen. Mit ihrer Hilfe kann die Modellierung plattformunabhängig erfolgen, indem funktionale und nicht-funktionale Anforderungen sowie Wichtungsregeln hinterlegt werden, die Grundlage für die kontextsensitive ✪ Discovery und Integration von Komponenten zur Laufzeit bilden. Aufgrund dieser ✪ Plattformunabhängigkeit ist das Modell konzeptionell nicht auf Webanwendungen beschränkt.

Die ✪ lose Kopplung von Komponenten erfolgt durch Verknüpfung von Ereignissen und Operationen von Komponenten über Links. Dabei wird den Anforderungen hinsichtlich der ✪ Koordination auf der UI-Ebene gesondert Rechnung getragen. So beschränken sich Links nicht auf die unidirektionale Weiterleitung von Daten, wie in bestehenden Ansätzen aus Forschung und Technik. Vielmehr bilden sie erweiterte Kommunikationsmuster, wie die aktive Abfrage von Daten, die asynchrone und dauerhafte Aktualisierung, die Synchronisation von Komponenten sowie plattformspezifische Techniken, wie Drag-and-Drop, auf das Event-Modell ab.

Zur Steigerung der ✪ Interoperabilität bietet das Modell mehrere Möglichkeiten: Logik-Komponenten und Parameter-Mappings erlauben die Überbrückung syntaktischer Differenzen auf Signaturebene, während die semantische Typisierung die spätere ✪ Mediation auf der Datenebene ermöglicht.

Den Anforderungen an die ✪ Präsentation wird durch Stile und Layouts Rechnung getragen. Stile repräsentieren visuelle Eigenschaften, die zur Homogenisierung der Oberfläche auf Komponenten und Layouts angewendet werden können. Durch Layouts und ihre hierarchische Schachtelung können UI-Komponenten nach Belieben auf der Oberfläche positioniert werden. Im Vergleich zu bestehenden Ansätzen, die sich i. d. R. auf HTML-Templates beschränken, zeichnet sich das Modell durch große Flexibilität und ✪ Plattformunabhängigkeit aus.

Schließlich erlaubt das Screenflow-Modell die Definition von Sichten auf eine komposite Anwendung. Jede davon repräsentiert die jeweils aktiven Komponenten und ihre Anordnung, die sich in Abhängigkeit von Events ändern können. Durch die ✪ Universalität besteht keine Beschränkung auf Nutzerinteraktionen bei Sicht- bzw. Seitenwechseln, wie häufig bei Web-Engineering-Ansätzen. Vielmehr können Nutzer-, Komponenten-, System- und Kontextereignisse gleichermaßen zum Wechsel beitragen. Als Ausgangspunkt für die ✪ Kontextsensitivität können zudem alternative Screenflows, z. B. gebunden an Geräte- oder Nutzerklassen, definiert werden.

Das Kompositionsmodell und das inhärente Komponentenmodell werden somit den gestellten funktionalen und nicht-funktionalen Anforderungen gerecht. Sie erlauben die Anwendung eines modellgetriebenen Entwicklungsprozesses, wie in Abschnitt 5.4 angedeutet, in Verbindung mit den in Abschnitt 2.1.3 geschilderten Vorteilen.

Modellierung von adaptivem Verhalten

Im dritten Teilkapitel 5.3 wurde die ✪Erweiterbarkeit des Kompositionsmodells (auf Metamodellebene) anhand des Adaptionsmodells gezeigt. Dieses ermöglicht die Modellierung von adaptivem Laufzeitverhalten bzw. ✪Kontextsensitivität einer kompositen Anwendung. Sie erfolgt unabhängig von den anderen Teilmodellen, d. h. es existieren keine direkten Abhängigkeiten und die ✪SoC bleibt gewahrt. Zudem steigt die Les- und Wartbarkeit von Adaptionslogik und die ✪Wiederverwendbarkeit in verschiedenen Kompositionen ist gegeben.
Die Repräsentation des Kontextmodells erfolgt ebenfalls als Komponente nach dem o. g. Modell – es wird somit inhärenter Teil einer Komposition und kann direkt mit anderen Komponenten verknüpft werden. Die Synchronisation von Komponenten- mit Kontexteigenschaften erfolgt über *Context Links*.
Darüber hinaus können Anpassungen der Komposition als Reaktion auf beliebige Ereignisse – i. d. R. Kontextänderungen, aber auch Anwendungs- und Systemereignisse – formuliert werden. Dazu wurden Adaptionsaspekte vorgestellt, die dynamische Änderungen am Kompositionsmodell repräsentieren. Sie ziehen die Umsetzung einer oder mehrerer der vorgestellten Adaptionstechniken (vgl. Abschnitt 5.3.1), nach sich, wie die Rekonfiguration oder den Austausch einzelner Komponenten sowie Änderungen des Daten- und Kontrollflusses, von Layouts oder Sichten. Ziel jeder Adaption sind somit Bestandteile des Kompositionsmodells, die direkt oder indirekt durch Mittel zur ✪Abstraktion adressiert werden können. Durch die Anwendung von Adaptionsaktionen auf (semantische) Klassen von Eigenschaften, Komponenten, Links, Layouts, usw. kann die Redundanz der Adaptionslogik vermindert und ihre Anwendungs- und ✪Plattformunabhängigkeit erhöht werden.
Aufgrund der ✪Komponentenorientierung findet die Adaption auf einer höheren Abstraktionsschicht statt, als bei *Adaptive-Hypermedia*-Ansätzen. Deren Techniken können komplementär betrachtet werden und innerhalb von Komponenten zum Einsatz kommen. Gegenüber vergleichbaren Konzepten aus den Bereichen CBSE und SOA adressiert die vorgestellte Lösung erstmals zustandsbehaftete Komponenten der Präsentationsebene.

Die in diesem Kapitel vorgestellten Modellierungsmittel decken somit alle geforderten Anforderungen ab und bilden die Grundlage für die angestrebte, plattformunabhängige Entwicklung kompositer Mashup-Anwendungen. Dazu sind verschiedene Vorgehensmodelle denkbar, die von der konventionellen, schrittweisen Entwicklung, wie in 5.4 skizziert, bis hin zur dynamischen Erstellung in Form von End-User-Development reichen. Zur Ausführung einer Komposition bedarf es einer Infrastruktur, die – ausgehend von dem Kompositionsmodell einer Anwendung – die kontextsensitive Suche, Integration und Komposition aller Komponenten sowie die Ausführung und Adaption der kompositen Anwendung realisiert. Das nächste Kapitel widmet sich der Vorstellung der entsprechenden Konzepte.

6

Kontextsensitiver Integrationsprozess und Kompositionsinfrastruktur für adaptive, interaktive Mashup-Anwendungen

Im vorigen Kapitel wurden die Konzepte zur Modellierung interaktiver Mashup-Anwendungen vorgestellt. Grundlage bildet ein universelles Komponentenmodell und eine entsprechende Beschreibungssprache, die verteilte Anwendungsbausteine deklarativ und unabhängig von Implementierungsdetails beschreibt. Auf Basis dieser Schnittstellenbeschreibung werden sie zu einer Anwendung komponiert, deren funktionale Zusammenhänge in einem Kompositionsmodell plattformunabhängig spezifiziert werden. Neben der Modellierung grundlegender Anwendungsbelange, wie Konfiguration, Daten- und Kontrollfluss oder Layout, wurden mit dem Adaptionsmodell Konzepte zur aspektorientierten Beschreibung von adaptivem bzw. kontextabhängigem Verhalten zur Laufzeit vorgestellt.

Das Anwendungs- bzw. Kompositionsmodell bildet die Basis für die modellgetriebene Entwicklung adaptiver, kompositer Webanwendungen. Für deren Bereitstellung bedarf es wissenschaftlicher Konzepte zur Verwaltung von Komponenten, zur Überwachung und Interpretation des Kontextes, zur kontextabhängigen Suche, Auswahl und Integration von Komponenten sowie zur Komposition und koordinierten Ausführung der Anwendung.

Diese Aufgaben stellen an die Systeminfrastruktur und Laufzeitumgebungen Herausforderungen, für die im Rahmen dieser Arbeit neue, wissenschaftliche Konzepte erarbeitet wurden. Sie dienen u. a. der dynamischen Suche, Auswahl und Einbindung von Komponenten auf Basis der im letzten Kapitel beschriebenen Abstraktionen im Kompositionsmodell, ihrer losen Kopplung und der Ausführung sowie kontextabhängigen Adaption der kompositen Anwendung entsprechend der modellierten Adaptionsaspekte. Zu diesem Zweck wurden eine serviceorientierte Infrastruktur sowie verschiedene Laufzeitumgebungen geschaffen, um die Plattformunabhängigkeit der Modellierung zu untermauern.

Dieses Kapitel widmet sich der detaillierten Darstellung der wissenschaftlichen Konzepte, die in Abschnitt 4.3 bereits überblicksartig vorgestellt wurden. Zunächst beschreibt Abschnitt 6.1 die Teilschritte der kontextsensitiven Integration von Mashup-Komponenten, ausgehend von der Modellinterpretation bis zur Einbindung der dynamisch gesuchten und gebundenen Komponenten in eine komposite Anwendung. In Abschnitt 6.2 liegt der Schwerpunkt der Betrachtung auf Ausführungsumgebungen für komposite Mashup-Anwendungen. Es werden die funktionalen Module einer Laufzeitumgebung für universelle Kompositionen vorgestellt und ihr Zusammenspiel erläutert. Besonderes Augenmerk gilt der Umsetzung des Daten- und Kontrollflusses unter Berücksichtigung der losen Kopplung und möglichst hohen Interoperabilität von Komponenten. Der Abschnitt widmet sich deshalb insbesondere den Kommunikations- und Mediationskonzepten. Abschnitt 6.3 stellt schließlich die Adaptionskonzepte vor. Dabei wird auf die Infrastruktur zur Modellierung und Folgerung von Kontextwissen, sowie deren Nutzung zur Umsetzung der modellierten Adaptionsbelange eingegangen. Das Kapitel schließt mit einer Zusammenfassung und Diskussion der vorgestellten Konzepte.

6.1 Ein kontextsensitiver Integrationsprozess zur dynamischen Bindung von Mashup-Komponenten

Grundlage der kontextadaptiven, dynamischen Komposition von Anwendungen auf Basis des im letzten Kapitel beschriebenen Modells bildet der im Folgenden vorgestellte Integrationsprozess. Abbildung 6.1 verdeutlicht die Abfolge der einzelnen Phasen bzw. Aufgaben (grün hinterlegt), den Bezug zu den im letzten Kapitel vorgestellten Modellen (grau) sowie die entsprechenden Infrastrukturkomponenten (blau im Hintergrund), auf die später eingegangen wird. Prinzipiell ähnelt der Ablauf von Ansatz und Struktur dem SWS Nutzungsprozess aus Abschnitt 3.1.2, auf dessen Beschränkungen hinsichtlich des Komponentenmodells, Kompositionstyps, usw. bereits an dieser Stelle eingegangen wurde. Ausgehend vom ① Zielmodell einer kompositen Anwendung (*composition model*) werden mehrere Phasen durchlaufen, in denen ② das Modell interpretiert, ③ funktional und semantisch kompatible Komponenten gesucht (*Matching*), ④ bezüglich ihrer Eignung im Kontext der Nutzung sortiert (*Ranking*), ausgewählt und ⑤ letztlich integriert sowie initialisiert werden (*Binding*). Zur Laufzeit können syntaktische Differenzen durch ⑥ Mediationsverfahren ausgeglichen werden.

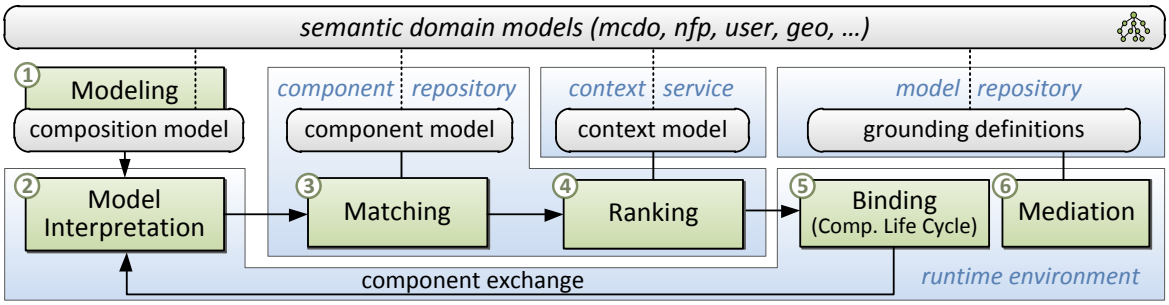

Abb. 6.1: Phasen und Module im dynamischen Integrationsprozess für Mashup-Komponenten

Die Integration „konkreter“ Mashup-Komponenten, die über ihre ID eindeutig im Kompositionsmodell referenziert sind, verläuft direkt: Die Suche beschränkt sich auf die Abfrage der Komponente über ihre ID und die kontextabhängige Sortierung möglicher Alternativen entfällt. Jede konkrete Komponente kann, z. B. bei Problemen zur Initialisierungs- oder Laufzeit, jedoch auch als Template aufgefasst werden. Dies führt zur Suche nach kompatiblen Alternativen, die im Folgenden genauer beschrieben wird. Die dynamische Bindung von Komponenten auf Basis semantischer Abstraktionen dient somit sowohl der Plattformunabhängigkeit im Entwicklungsprozess als auch der Fehlerbehandlung zur Laufzeit.
In den folgenden Abschnitten wird dieser kontextsensitive Integrationsprozess von der Modellinterpretation bis zur Bindung der Komponenten ausführlich erläutert.

6.1.1 Modellinterpretation oder -transformation

Den Ausgangspunkt der dynamischen Integration und Komposition von Mashup-Komponenten stellt ein Zielmodell der Anwendung dar. Je nach Mächtigkeit der Laufzeitumgebung kann entschieden werden, ob zunächst eine Transformation des Modells in eine ausführbare Anwendung vorausgeht, oder das Modell direkt interpretiert wird. Im Rahmen dieser Arbeit wurden beide Wege exemplarisch umgesetzt. In jedem Fall werden die Komponenten erst mit der Initialisierung der Anwendung ausgewählt und geladen – bei vorheriger Transformation muss ein Konzept für Platzhalter existieren, die bei der Initialisierung ersetzt werden.
Als Grundlage für die weitere Beschreibung soll ein Beispiel dienen, welches in den nächsten Abschnitten immer wieder aufgegriffen wird.

Beispiel: *Das Kompositionsmodell der in Abschnitt 2.2.1 skizzierten Reiseplanungsanwendung beinhaltet eine Komponente zur Darstellung von Objekten des Typs EventLocation (Konzertsäle, Clubs, Bars, etc.). Sie ist durch ein Template T modelliert, um die Entscheidung bzgl. ihrer Darstellungsform (Karte, Liste, usw.) und Technologie (JavaScript, Flash, Silverlight, usw.) auf die Erfordernisse und Möglichkeiten zur Laufzeit abstimmen zu können. T besitzt u. a. eine Operation setMarker(EventLocation marker), über die neue Orte hinzugefügt werden können.*

Für jedes Template im Conceptual Model (Abschnitt 5.2.1) wird bei der Modellinterpretation der Integrationsprozess angestoßen. Dazu erfolgt eine Anfrage an das Komponenten-Repository (CoRe) mit den folgenden Informationen: (1) Das Template selbst, d. h. die definierte Schnittstelle samt funktionaler und Datensemantik sowie anzuwendender Sortierungsregeln; (2) Plattform-Identifier; (3) Nutzer-Identifier, z. B. die ID des Nutzers zur Auflösung des Kontextmodells; (4) zusätzliche optionale Parameter, wie die Anzahl gewünschter Komponentenvorschläge und eine *Blacklist*.
Die Angabe des Plattform-Identifiers setzt voraus, dass die Plattform in der semantischen Datenbasis des Repositorys hinsichtlich ihrer funktionalen und nichtfunktionalen Eigenschaften beschrieben wurde. Alternativ ist es möglich, direkt das entsprechende semantische Modell der Plattformcharakteristika zu übergeben. In Anbetracht des Einsatzes einer begrenzten Anzahl fest definierter Plattformen sowie aus Gründen der Praktikabilität wird empfohlen, lediglich eine ID zu übergeben. Analog verhält es sich mit den Nutzer-Identifier. Es wird unterstellt, dass für jeden Nutzer ein Modell verwaltet wird, auf welches Laufzeitumgebung und CoRe Zugriff haben.

Die Authentifizierung kann extern über Mechanismen wie OAuth [@Hammer-Lahav, 2010] oder OpenID [@OpenID, 2007] erfolgen, sodass eine einheitliche Zuordnung des Nutzers in der gesamten Kompositionsinfrastruktur möglich ist. Durch die Referenzierung des Nutzers erfolgt gleichsam die Zuordnung der entsprechenden Kontexteigenschaften beim Kontextdienst (vgl. Abschnitt 6.3.1) im Rahmen der kontextadaptiven Suche. Mit der Blacklist können schließlich Komponenten von der Suche ausgeschlossen werden, z. B. bei der Suche nach Alternativen.
Neben der „Auflösung" der Templates erfolgt die Interpretation und Abbildung der spezifizierten Anwendungslogik in Form von Layout, Screenflow sowie Kontroll- und Datenfluss auf die Mechanismen der Laufzeitumgebung. Da diese Abbildung je nach Plattform variiert, soll sie hier nicht näher thematisiert werden. Beispiele für die Möglichkeiten der Interpretation können Abschnitt 7.2.2 entnommen werden.
Mit der Anfrage nach Komponenten an das CoRe wird die Suche nach denjenigen Komponenten angestoßen, die bezüglich ihrer Schnittstellenbeschreibung funktional und bezüglich der Datensemantik mit der Vorlage aus dem Kompositionsmodell kompatibel sind. Dieser *Discovery*-Prozess besteht aus drei Schritten: (1) dem Abgleich der funktionalen Schnittstellen (*Matching*), (2) der Bewertung und Sortierung der Kandidatenmenge (*Ranking*) sowie (3) der Selektion von Komponenten. Der Ablauf dieser drei Schritte wird in den folgenden Abschnitten erläutert.

6.1.2 Suche und Matching

Der erste Schritt der Discovery beinhaltet – neben einer Vorselektion – im Wesentlichen das *Matching* aller funktionalen Schnittstellenbestandteile zwischen übermittelter Vorlage und registrierten Komponenten. Dabei handelt es sich prinzipiell um eine boolesche Entscheidung: entweder passt eine Komponente *exakt*, oder nicht. Durch die Nutzung semantischer Beschreibungen ist es jedoch möglich, Querbezüge und Vererbungen im semantischen Modell in den Vergleich einzuschließen. Somit können Komponenten auch dann integriert werden, wenn ihre Schnittstellen syntaktisch zwar differieren, über ihre Semantik jedoch ineinander überführt werden können.
Da die Discovery zur Lauf- bzw. Initialisierungszeit der Anwendung stattfindet, können nur vollständig kompatible Komponenten zurückgeliefert werden. Nichtlogische und hybride Suchverfahren (vgl. Abschnitt 3.1.2) scheiden somit aus, da die Komposition im Falle partieller oder ungenauer Treffer nicht mehr durch Entwickler angepasst werden kann. Abweichungen zwischen Template und Kandidat sind nur soweit tolerierbar, wie sie durch semantische Mediation später automatisch überbrückbar sind.
Für das Matching im Rahmen des vorgeschlagenen Integrationsprozesses wurde ein logikbasierter Algorithmus definiert, welcher Subsumptionsbeziehungen zwischen semantischen Anfragen (Schnittstelle I_t der *Templates*) und Angeboten (Schnittstelle I_c registrierter Komponenten) ausnutzt und eine Liste von *Mappings* zwischen diesen zurückliefert. Der Algorithmus selbst ist generisch und von keiner der genutzten Domänenontologien abhängig.
Vor der Anwendung des Matching-Algorithmus auf die Kandidatenmenge wird diese zunächst, wie in YASA-M [Chabeb et al., 2010] vorgeschlagen, eingegrenzt. Kriterien für die **Vorselektion** bieten die Informationen aus der Anfrage durch die Laufzeitumgebung. Zunächst können alle Komponenten von der mitgelieferten Blacklist

ausgeschlossen werden. Weiterhin entfallen Komponenten, die die Zielplattform technologisch nicht unterstützen. Dies kann durch den Abgleich der Plattformcharakteristika mit den nicht-funktionalen Informationen der SMCDL geschlossen werden (vgl. Abschnitt 5.1.3.3). Die Analyse der minimalen Anzahl benötigter Eigenschaften, Ereignisse und Operationen laut Vorlage führt zum Ausschluss aller Komponenten, die „weniger" bieten und somit nicht für die Vorlage eingesetzt bzw. mit den Kommunikationskanälen verbunden werden können. Im Gegensatz zum Vorgehen von Chabeb et al. (2010) werden allerdings Komponenten mit mehr Operationen und Ereignissen zum Matching zugelassen, da sie die geforderte Funktionalität voll abdecken können. Der tatsächliche Grad der funktionalen Abdeckung und interne Abhängigkeiten werden später durch den Algorithmus berücksichtigt. Die Vorselektion bezieht außerdem die Art der geforderten Komponente ein. Service-Komponenten fallen bei der Anfrage nach UI-Komponenten beispielsweise aus der Kandidatenmenge (angezeigt durch das entsprechende Attribut der MCDL).
An die Vorselektion schließt sich die Ermittlung der semantischen Kompatibilität der Vorlage mit Komponenten der Kandidatenmenge an. Im Folgenden werden zunächst Grundlagen des eingesetzten Matching-Algorithmus und dann dessen Anwendung im Rahmen der dynamischen Integration erläutert.

6.1.2.1 Theoretische Grundlagen

Grundlage des Algorithmus bilden die semantischen Annotationen der SMCDL sowie die semantische Typisierung von Parametern und Eigenschaften der Komponenten. Auf Basis der referenzierten Konzepte ist es möglich, den Übereinstimmungsgrad $AtomicMatch$ zwischen Schnittstellenbestandteilen einer Vorlage und einer Komponente zu vergleichen. Die Berechnung erfolgt in Anlehnung an die in Abschnitt 3.1.2 vorgestellten logikbasierten Verfahren für SWS nach folgendem Schema:

$$AtomicMatch(C_1, C_2) = \begin{cases} Exact & , wenn\ C_1 \equiv C_2 \\ Plugin & , wenn\ C_1 \sqsubseteq C_2 \\ Subsumes & , wenn\ C_1 \sqsupseteq C_2 \\ Fail & ansonsten \end{cases}$$

Das $AtomicMatch$ ergibt sich aus dem subsumptionsbasierten Matching, definiert über *subClassOf*. Neben der exakten Übereinstimmung (*Exact*) der Konzepte werden auch die Grade *Plugin* und *Subsumes* als Teiltreffer unterstützt. Der Unterschied zwischen beiden wird am Beispiel deutlich:

Beispiel: *Gegeben sei eine Vorlage T, die eine Operation mit einem Parameter des Typs Location aufweist. Dieser Typ wird im semantischen Modell – z. B. formuliert in OWL DL – durch das Konzept EventLocation spezialisiert und um eine Eigenschaft hasEvent erweitert, d. h. es besteht eine Subklassenbeziehung. Eine Komponente C mit der Operation setLocation(EventLocation) steht folglich in einer* Subsumes*-Beziehung zu T. Ihre Operation kann nicht ohne weiteres alle Eingaben aus der Komposition verarbeiten, da sie u. U. auf die Informationen aus hasEvent angewiesen ist. Im umgedrehten Fall – T verlangt EventLocation und C unterstützt Location – herrscht ein* Plugin*-Verhältnis.*

Die Interpretation von $AtomicMatch$ ist abhängig vom betrachteten Schnittstellenbestandteil. Das obige Beispiel zeigt, dass der Übereinstimmungsgrad *Subsumes* beim

Abgleich der Datensemantik nicht unterstützt werden kann, da die Verarbeitung der Eingabe durch die Zielkomponente nicht sichergestellt werden kann. Beim Vergleich funktionaler Semantik kann die Spezialisierung jedoch durchaus Relevanz besitzen: Bietet eine Komponente das funktionale Konzept SecureLogin, so erfüllt sie die Anforderung Login einer Vorlage vollständig und kommt als Kandidat in Frage.
In diesem Zusammenhang spielt die in Abschnitt 3.1.2 beschriebene *Contravariance* eine wichtige Rolle. Im Bezug auf das vorgestellte Komponentenmodell sagt sie aus, dass sich die atomaren Übereinstimmungsgrade zwischen Vorlagen und Komponenten für Parameter von Events ($AtomicMatch(type_c, type_t)$) und Operations ($AtomicMatch(type_t, type_c)$) genau invers verhalten. Dies ist der Tatsache geschuldet, dass in Richtung des Datenflusses immer nur *Up-Casts* möglich sind, damit verarbeitenden Instanzen alle nötigen Informationen zur Verfügung stehen. Event-Parameter von Kandidaten müssen somit den gleichen oder einen spezielleren Typ als den in der Vorlage definierten aufweisen – Parameter der Operationen von Kandidaten müssen hingegen gleiche oder allgemeinere Typen aufweisen.

Beispiel: *Das Template T aus Abbildung 6.2 verarbeitet in einer Operation den über einen Link (vgl. Abschnitt 5.2.2) bereitgestellten Parameter vom Typ EventLocation. Zwei Kandidaten C_1 und C_2 bieten Operationen an, die alle Anforderungen der Vorlage erfüllen. Die Operation von C_1 kann den Parameter direkt verarbeiten – für C_2 muss zunächst ein Up-Cast in das Superkonzept Location erfolgen.*
T besitzt weiterhin ein Event, welches einen Parameter vom Typ EventLocation publiziert. Dieser kann jedoch nur von C_1 durch einen Up-Cast aus der Subklasse ConcertLocation bereitgestellt werden, denn dem Location-Parameter von C_2 fehlen die potentiell nötigen Event-spezifischen Informationen. Komponente C_1 ist somit der einzig valide Kandidat, der die Anforderungen von T hinsichtlich der Datenschnittstelle erfüllt, während C_2 im Matching ausgeschlossen wird.

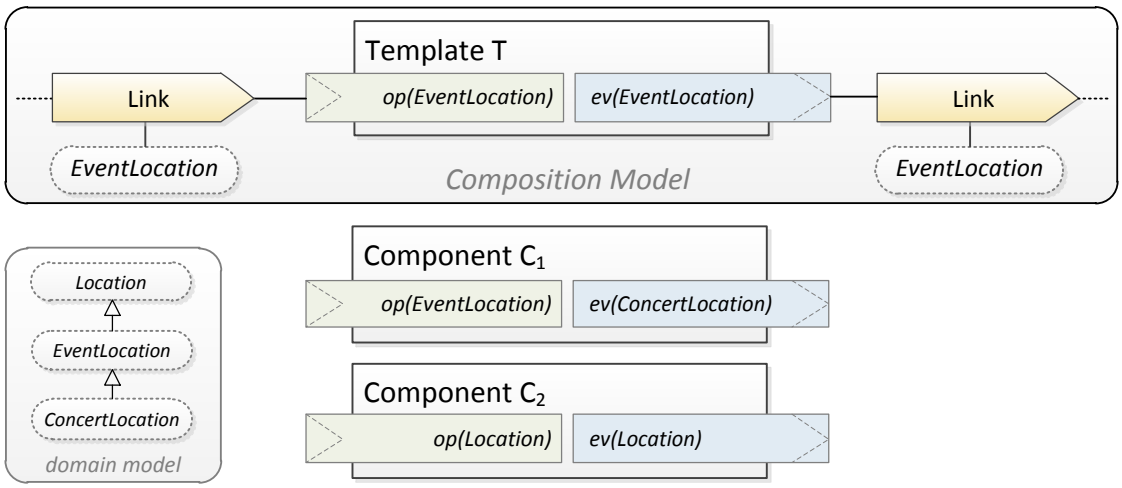

Abb. 6.2: Einfluss der Contravariance auf das Matching

Das Beispiel verdeutlicht, dass beim Matching von Events die Parameter der Kandidaten durch Superkonzepte der Vorlage typisiert sein können, während beim Vergleich der Operationen nur Spezialisierungen unterstützt werden können. In Anbetracht dieser Besonderheit muss die Übereinstimmung für Ereignisse und Operationen invers bewertet werden:

Operation: $AtomicMatch(C_{template}, C_{component})$
Ereignis: $AtomicMatch(C_{component}, C_{template})$

Das Ergebnis des Vergleiches wird in numerischer Form repräsentiert ($Exact = 3, PlugIn = Subsumes = 1, Fail = 0$), was die einfache Aggregation der Matchinggrade aller Schnittstellenbestandteile ermöglicht. Die exakte Übereinstimmung wird höher bewertet, da die Nutzung der entsprechenden Konzepte zur Laufzeit keinerlei Mediation (vgl. Abschnitt 6.2.4.2) bedarf. Dem erhöhten Aufwand der Überführung semantischer Datentypen durch Casts zur Laufzeit wird im Falle von *PlugIn* und *Subsumes* durch niedrigere Werte Rechnung getragen. Da dieser Aufwand stark von den genutzten Domänenmodellen und überspannten Hierarchiestufen abhängig ist, lässt er sich nicht allgemeingültig quantifizieren (z. B. durch höhere oder niedrigere Wichtung in Relation zu *Exact*). Ebenso lässt sich keine allgemeingültige Aussage treffen, ob *PlugIn* oder *Subsumes* aufwändiger ist, weshalb beide Matchinggrade gleich gewertet werden.

6.1.2.2 Matching-Algorithmus

Der im Rahmen dieser Arbeit konzipierte Algorithmus setzt die o. g. Prinzipien um. Als Eingabe erhält er eine Vorlage aus dem Kompositionsmodell, für die er eine Liste semantisch kompatibler Komponenten, deren Übereinstimmungsgrad und Abbildungsvorschriften auf das geforderte Template zurückliefert. Intern arbeitet er auf einer semantischen Wissensbasis, die alle registrierten Mashup-Komponenten mit Hilfe der MCDO und entsprechenden Domänenontologien repräsentiert. Die Funktionsweise des Algorithmus wird im Folgenden näher erläutert.
An die oben beschriebene Vorselektion, die im Wesentlichen syntaktische Aspekte berücksichtigt, schließt sich das semantische *Matchmaking* in vier Schritten für (1) die Grundfunktionalität, (2) die Eigenschaften, (3) Operationen und (4) Ereignisse der Vorlage an.

Matching der Grundfunktionalität

In einem ersten Schritt wird geprüft ob die Kandidaten den funktionalen Anforderungen gerecht werden, die auf der Komponentenebene semantisch annotiert sind. Die referenzierten Konzepte drücken i. d. R. Funktionalitäten im Sinne einer Aufgabe eines Task-Modells aus, z. B. *Routenplanung*. Der Vergleich erfolgt paarweise zwischen Vorlage ($func_t$) und allen Kandidaten ($func_c$) auf Basis von $AtomicMatch$. Die Einzelwerte des Vergleichs werden gespeichert und dienen im Anschluss der Ermittlung der besten Zuordnung – vergleichbar mit dem Vorgehen in SAWSDL-MX [Klusch et al., 2009]. Kann ein $func_c$ nicht zugewiesen werden, d. h. alle paarweisen Vergleiche mit den Annotationen von C resultieren in *Fail*, so wird der Kandidat vom weiteren Matching ausgeschlossen. Wurde für alle obligatorischen funktionalen Konzepte eine Zuordnung gefunden, so wird der Übereinstimmungsgrad $match_{funct}$ als Mittelwert der numerischen Werte aus dem atomaren Matching ermittelt. Bei der Aggregation sind für nicht-optionale Funktionalitäten die Gewichtungen aus der Vorlage entsprechend zu beachten.
Eine Komponente, die alle obligatorischen funktionalen Annotationen aufweisen kann, die durch die Vorlage gefordert werden, erhält somit die Wertung 3. Einer Komponente, deren funktionale Annotationen alle Plugin- oder Subsumes-Matches

darstellen, wird $match_{funct} = 1$ zugewiesen. Tabelle 6.1 illustriert die Berechnung anhand einer Vorlage mit vier funktionalen Modellreferenzen und zwei Kandidaten. In diesem Fall muss C_2 trotz seines niedrigeren Übereinstimmungsgrades gewählt werden, da C_1 das vierte benötigte Konzept nicht unterstützt.

Gewichtung der Modellreferenzen					
Template	1.0	1.0	0.5	0.3	
	Übereinstimmungsgrad der Kandidaten				$match_{funct}$
Kandidat C_1	3	3	3	0	**~~2.68~~**
Kandidat C_2	1	3	3	3	**2.29**

Tab. 6.1: Beispiel der Aggregation elementarer Übereinstimmungen zu $match_{funct}$

Matching von Properties

Im nächsten Schritt erfolgt die Ermittlung der bestmöglichen Zuordnung jeder Eigenschaft einer Vorlage zu einer semantisch passenden Eigenschaft eines Kandidaten. Wiederum wird paarweise der Deckungsgrad $match_{prop}$ durch $AtomicMatch$ anhand der semantischen Typisierung der jeweiligen Property berechnet. Die beste Zuordnung zwischen Vorlage und Kandidat wird als Tupel der entsprechenden Properties und ihrem Übereinstimmungsgrad gespeichert ($bestAssign_{prop} = \{\langle prop_t, prop_c, rating\rangle \mid prop_t \in I_t, prop_c \in I_c\}$). Findet sich für eine Eigenschaft der Vorlage keine passende Property beim Kandidaten, so wird dieser für das weitere Matching verworfen. In Abbildung 6.3 ist dies für Komponente C_1 der Fall, da sie für die geforderte Eigenschaft „Destination“ kein Äquivalent bietet. C_2 bietet indes zwei exakt typgleiche Eigenschaften und verbleibt folglich in der Kandidatenmenge.

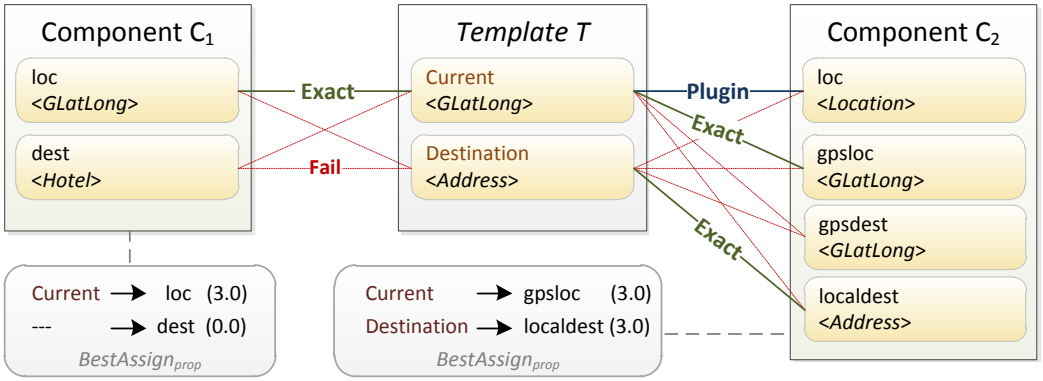

Abb. 6.3: Matching zweier Komponenten bezüglich geforderter Properties

Im Gegensatz zum Vergleich der funktionalen Annotationen sind hier lediglich *exakte* Übereinstimmungen zulässig, da Properties eine Schlüsselrolle beim Komponententausch zukommt. Dabei werden die Werte aller zustandsgebenden Eigenschaften von der zu ersetzenden auf die neue Komponente übertragen. Bei einem Übereinstimmungsgrad von *Plugin* für das Property einer initialen Komponente A kann dessen Wert laut Kompositionsmodell zwar auf den semantischen Typ von A gecastet werden, z. B. von ConcertLocation auf EventLocation – beim Austausch mit einer vorlagengerechten Komponente B ist es jedoch nicht möglich, den ausgelesenen Wert (EventLocation) zurück in eine Subklasse (ConcertLocation) zu überführen.

Durch die exakte Übereinstimmung ist keine Mediation bei der Kommunikation über Property-Links nötig.
Ein Problem bei der Ermittlung von $bestAssign_{prop}$ stellen mehrere semantisch gleiche Properties dar, da sie die eindeutige Zuordnung durch den Algorithmus verhindern. Zu diesem Zweck können – in Anlehnung an die hybriden Matching-Verfahren aus Abschnitt 3.1.2 – nicht-logische Verfahren zum Einsatz kommen, beispielsweise die Berücksichtigung textueller Ähnlichkeit oder die Einbeziehung von Wörterbüchern und Thesauri. Diese Verfahren sind jedoch inhärent unsicher, sodass bei der Entwicklung von Komponenten auf eine möglichst konkrete Typisierung, im Zweifelsfall durch Spezialisierung der Konzepte, geachtet werden sollte.

Matching von Operationen

Der nächste Schritt umfasst die Suche möglicher Abbildungen von Operationen der Vorlage op_t auf jene der Kandidaten op_c, wie in Abbildung 6.4 verdeutlicht. Dieser Vorgang ist zweistufig, da die Operationen sowohl bezüglich ihrer Funktionalität als auch bezüglich der Anzahl und semantischen Typisierung der ausgetauschten Parameter kompatibel sein müssen. In Hinblick auf den letzteren Punkt können zunächst alle Operationen für das Matching entfallen, die mehr Parameter als durch die Vorlage gefordert verlangen. Operationen mit der gleichen oder einer geringeren Anzahl bleiben mögliche Kandidaten, sofern sie die gewünschte Funktionalität auch mit ggf. weniger Parametern ermöglichen. Dies ist i. d. R. der Fall, wenn die Vorlage optionale und redundante Parameter enthält, um die Treffermenge zu erhöhen.
Die Überprüfung der **funktionalen Semantik** von Operationen basiert auf den daran annotierten Modellreferenzen unter Beachtung ihrer optionalen Gewichtung (vgl. Abschnitte 5.1.3.3 und 5.1.3.4 zur Beschreibung funktionaler Semantik). Vor dem Abgleich müssen zunächst etwaige Vorbedingungen überprüft werden, wie sie als Teil der MCDO formulierbar sind. Sind sie erfüllt, wird $match_{funct}$ analog zur Grundfunktionalität der Komponente ermittelt. Kann für eine Modellreferenz der Vorlagen-Operation beim Kandidaten keine Zuordnung gefunden werden, bricht das Matching für diese Operation ab und resultiert in *Fail*.
Für die verbleibenden Operationen folgt das Matching der **Datensemantik**. Jeder Parameter von op_c muss zumindest in einer *PlugIn*-Beziehung zu einem Parameter von op_t stehen. Dadurch ist gewährleistet, dass alle zum Aufruf nötigen Parameter vorhanden sind und kompatible Datentypen aufweisen. Kann keine derartige Beziehung gefunden werden, wird die Übereinstimmung mit *Fail* bewertet. Für die übrigen Operationen wird der Übereinstimmungsgrad der Parameter $match_{param}$ von op_c aus dem Durchschnitt aller atomaren Matches ermittelt.

Beispiel: *Abbildung 6.4 veranschaulicht das Matching der Operationen zwischen Vorlage T und Komponente C_2. Aus Sicht der Komposition wird eine Operation* update *benötigt, die über einen Parameter vom Typ* EventLocation *verfügt. Die erste Operation von C_2 wird ausgeschlossen, da für ihren Aufruf nicht alle nötigen Parameter verfügbar sind. Die Operation* showEvent *bietet zwar eine funktionale Spezialisierung der gewünschten Funktionalität, allerdings keine kompatiblen Parameter. Die Operation* reset *kann potentiell aufgerufen werden, passt aber aufgrund ihrer funktionalen Annotationen nicht zur Vorlage. Mit der letzten Operation* setMarker *können schließlich alle Anforderungen aus T erfüllt werden. Bei der Nutzung von C_2 wird dann die Mediation des Parameters nötig, da lediglich ein* Plugin*-Match vorliegt.*

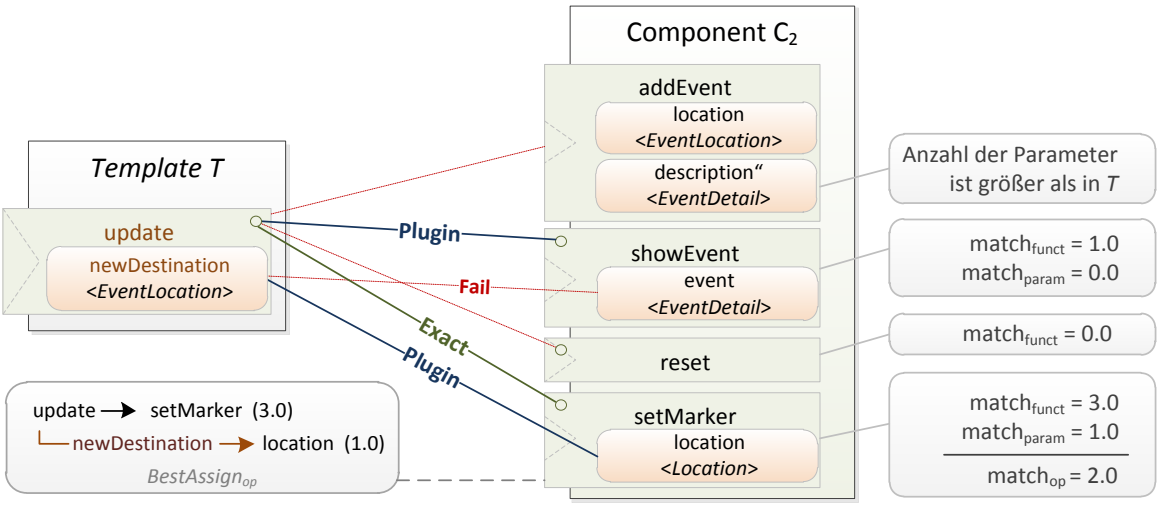

Abb. 6.4: Matching von Operationen zwischen Vorlage und Kandidat

In die Berechnung der Passgenauigkeit $match_{op}$ einer Komponentenoperation gehen $match_{funct}$ und $match_{param}$ zu gleichen Teilen ein. Durch die Mittlung über alle funktionalen Referenzen und Datentypen bleibt das Ergebnis unabhängig von der Anzahl der betrachteten Konzepte und Parameter. Unter allen möglichen Abbildungen zwischen Template und Komponente wird die beste Zuordnung $bestAssign_{op}$ bestimmt und gesichert. Sie repräsentiert jeweils ein Tupel aus den Operationsnamen der Vorlage und der Komponente, $match_{op}$ sowie der individuellen Parameterzuordnung. Kann die Operation aus T keiner Operation der untersuchten Komponente zugewiesen werden, so entfällt die letztere als Kandidat für das weitere Matching.

Matching von Events

Zu guter Letzt erfolgt der paarweise Vergleich zwischen geforderten Events ev_t aus der Vorlage und Events ev_c der Kandidaten. Dieser Schritt verläuft analog zum Matching der Operationen und mündet in die Gesamtwertung $bestAssign_ev$. Allerdings verhält sich die Vorselektion bezüglich der Parameteranzahl genau anders herum, als bei Operationen: ev_c darf potentiell mehr Parameter besitzen, als von ev_t gefordert, da der Kandidat dann alle für die Komposition nötigen Parameter liefert und zusätzliche problemlos durch die Laufzeitumgebung ignoriert werden können.

Neben dem Matching werden die Abhängigkeiten zu Operationen geprüft (vgl. Abschnitt 5.1.3.3 zum Attribut *trigger*). Hintergrund ist, dass jene Operationen durch die Komponenten gebunden sein sollen, die für benötigte Events den Auslöser darstellen. Ist dies nicht der Fall, so stehen die Events der komponierten Anwendung nicht zur Verfügung, da ihre Auslöser nie aufgerufen werden.

Berechnung der Gesamtbewertung

Nach Abschluss der beschriebenen Vergleiche enthält die Kandidatenmenge nur noch kompatible Komponenten, d. h. für jede Property, jede Operation und jedes Event der Vorlage existiert eine Zuordnung zu entsprechenden Schnittstellenteilen jedes Kandidaten. Die Gesamtbewertung $match_{component}$ spiegelt für jede einzelne Komponente ihre Passgenauigkeit bezüglich des vorgegebenen Templates wider. Sie

ergibt sich durch folgende Formel[1]:

$$match_{component} = \frac{match_{funct} + \nu_p * \sum_{i=1}^{|BA_{prop}|} BA_{prop,i}.rating + \nu_o * \sum_{j=1}^{|BA_{op}|} BA_{op,j}.rating + \nu_e * \sum_{k=1}^{|BA_{ev}|} BA_{ev,k}.rating}{1 + \nu_p * |BA_{prop}| + \nu_o * |BA_{op}| + \nu_e * |BA_{ev}|}$$

Der Deckungsgrad zwischen Komponente und Vorlage entspricht somit dem Mittel aus ihrer funktionaler Übereinstimmung sowie der Passgenauigkeit ihrer Properties, Events und Operations. Alle vier Faktoren können durch die Belegung von ν verschieden gewichtet werden.
Tabelle 6.2 verdeutlicht die Berechnung anhand der in den Beispielen vorgestellten Vorlage T und Komponente C_2. Neben den Zuordnungen und Übereinstimmungsgraden aus Tabelle 6.1 sowie Abbildung 6.3 und 6.4 wird eine Teilübereinstimmung für ein gefordertes Event $locationSelected(Location)$ angenommen.

Zuordnung	*Name in T*	*Name in C_2*	*Übereinstimmungsgrad*
$match_{funct}$			2.29
$bestAssign_{prop}$	Current	gpsloc	3.00
	Destination	localdest	3.00
$bestAssign_{op}$	update	setMarker	2.00
$bestAssign_{ev}$	locationSelected	newSelection	1.80
$match_{component}$			**2.49**

Tab. 6.2: Berechnung von $match_{component}$ am Beispiel

Neben $match_{component}$ wird für jede Komponente das *MatchingResult* gebildet. Dieses Objekt enthält alle Abbildungs- bzw. Zuordnungsregeln, die später für die Integration der Komponente bzw. die Konfiguration von Laufzeitumgebung und Mediationsmechanismus nötig sind (vgl. Abschnitt 6.2.4.2).

Als Ergebnis der Matching-Phase liegt eine Liste von Komponenten vor, die bezüglich der funktionalen und Datensemantik zu einer gegebenen Vorlage kompatibel sind. Die Komponenten können anhand ihrer Passgenauigkeit sortiert werden, die in Form von $match_{component}$ numerisch vorliegt.
Im nächsten Schritt werden nicht-funktionale Kriterien in die Komponentensuche einbezogen, was die Zahl der Kandidaten weiter eingrenzt und ggf. die Sortierung beeinflusst. Der folgende Abschnitt beschreibt das zugrunde liegende Vorgehen.

6.1.3 Rangfolgebildung

Analog zum SWS *usage process* (vgl. Abschnitt 3.1.2) schließt sich an das funktionale Matching das *Ranking*, d. h. die Rangfolgebildung der syntaktisch und semantisch mit der Vorlage kompatiblen Komponenten an. Diese werden anhand kontextspezifischer und nicht-funktionaler Eigenschaften (NFP) sortiert und ggf. aus der weiteren Betrachtung ausgeschlossen. Die Entscheidung darüber wird aufgrund von Ranking-Regeln (*Ranking Rules*, vgl. Abschnitt 5.2.1.1). getroffen.
Während Templates im Kompositionsmodell der Zielbeschreibung gemäß SWS entsprechen, stellen diese Regeln die Sortierungsanweisungen bezüglich spezifischer

[1] $BA_{x,n}.rating$ entspricht jeweils der Bewertung ($rating$) des n'ten Schnittstellenartefakts vom Typ $x \in \{prop; op; ev\}$ aus der o. g. besten Zuordnung $BestAssign$.

nicht-funktionaler Eigenschaften dar. Sie können explizit als Teil einer Komposition modelliert werden, es besteht aber auch die Möglichkeit, bestehende Regeln aus dem Modell zu referenzieren. Da sie sich nur auf semantische, nicht-funktionale Konzepte beziehen, sind sie ausreichend generisch und anwendungsunabhängig, um in verschiedensten Kompositionen Verwendung zu finden. Ihre Auswahl und Einbindung in Kompositionen durch den *UI Mashup Composer* kann als Integration nicht-funktionaler Aspekte verstanden werden.
Grundlage für die Bewertung und (Aus-)Sortierung von Komponenten bilden die nicht-funktionalen Eigenschaften der SMCDL und die entsprechenden Konzepte der in Abschnitt 5.1.3.3 angesprochenen NFP-Ontologie. Der Ranking-Algorithmus verarbeitet also zwei Eingaben: (1) Die aus dem Matching resultierende Kandidatenmenge samt $match_{component}$ und deren nicht-funktionale Eigenschaften sowie (2) die Vorlage aus dem Kompositionsmodell samt aller für sie relevanten Ranking-Regeln. Zur Auswertung von Kontextbezügen wird zudem das Kontextmodell des entsprechenden Nutzers benötigt, auf welches durch die von der Laufzeitumgebung bereitgestellte Nutzerkennung zugegriffen werden kann.

6.1.3.1 Definition und Anwendung der Sortierungslogik

Durch Ranking-Regeln lässt sich die „Passgenauigkeit“ von Komponenten in einem bestimmten Kontext beschreiben. Sie dienen somit der Konfiguration des Sortierungsalgorithmus und bieten Entwicklern kompositer Anwendungen die Möglichkeit, die Auswahl von Komponenten zur Laufzeit zu beeinflussen.
Grundsätzlich besitzt jede Regel mindestens eine Bedingung, um nicht-funktionale Eigenschaften der Komponenten mit Kontextparametern oder Literalen abzugleichen. Zur Adressierung der NFP wird SPARQL genutzt, da es sowohl standardisiert als auch sehr ausdrucksstark ist. Es erlaubt die generische Abfrage von Daten eines RDF-Graphen – in diesem Fall der in Abschnitt 5.1.3.3 erwähnten, semantischen NFP-Konzepte. Für diese können über das SPARQL-Konstrukt `FILTER` Bedingungen angegeben und die Verknüpfung mit dem Kontextmodell über *Extension Functions* hergestellt werden.
Die Formulierung von SPARQL durch Nicht-Programmierer ist freilich nicht vorgesehen. Im Regelfall beschränkt sich die Discovery dann auf die funktionale Passgenauigkeit von Kandidaten. Anderenfalls muss Unterstützung durch das Autorenwerkzeug gegeben sein, die allerdings nicht im Mittelpunkt der Arbeit steht.
Der Ranking-Algorithmus unterstützt drei Regeltypen, die als Teil des Conceptual Models spezifiziert werden können, wie Abbildung 5.11 zeigt. Die folgenden Abschnitte stellen diese Typen und ihre Auswirkung auf die Rangfolgebildung vor.

Occurance Rules – Prüfung von Metadaten

Occurance Rules dienen der Überprüfung von Metadaten einer Komponente auf bestimmte Werte. Im Falle der Nichterfüllung kontextbezogener Anforderungen, z. B. beim Abgleich des Lizenzmodells oder Preises mit Präferenzen aus dem Nutzermodell, kann es zum Ausschluss von Komponenten kommen.
Codebeispiel 6.1 zeigt die Serialisierung einer Regel, die prüft, ob eine Komponente kostenlos ist. Hierzu werden – auf die Kandidatenmenge angewendet – zunächst alle Individuen der MCDO-Klasse Metadata abgefragt, die die Bedingung betrifft. Durch das Konstrukt `UNION` (Zeile 5) werden die verschiedenen Konzeptualisierungen

zur Preisangabe berücksichtigt, bevor über den FILTER-Ausdruck in Zeile 6 alle Individuen aussortiert werden, deren Preis höher als 0 ist. Exemplarisch wird danach die Einbindung von Kontextdaten – in diesem Fall der Standardwährung des Nutzers – über die queryContext-Erweiterung gezeigt. Nach der Filterung erfolgt die Zuordnung der Individuen von Metadata zu den entsprechenden Kandidaten. Das Ergebnis der Auswertung bildet die Menge der Komponenten, die konform zur Bedingung sind, wobei die Gewichtung aus dem Matching beibehalten wird.

```
1 <rankingRule xsi:type="ccm:OccuranceRule" id="maxPrice" isGlobal="true"
2     metadata="PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
3         SELECT ?metadata
4         WHERE {?metadata nfp:hasPricing ?p .
5             {?p nfp:hasValue ?v .} UNION {?p nfp:hasUpperBound ?v .}
6         FILTER (xsd:float(?v) <= 0.0 && mcdl:queryContext("/pers:hasCurrency/")) .}">
7 </rankingRule>
```

Lst. 6.1: Beispiel einer Occurance Rule

Normalize und Mapping Rules – Auswertung von Metadaten

Normalize Rules dienen der dynamischen Sortierung und Bewertung der Kandidatenmenge im Hinblick auf ein bestimmtes Metadatum mit numerischem Wertebereich. Dazu werden die entsprechenden Metadaten-Werte aller Komponenten normalisiert, d. h. der Bereich zwischen dem globalen Minimal- und Maximalwert wird auf den Bereich $[0, 1]$ abgebildet.
Codebeispiel 6.2 zeigt eine Regel zur Normalisierung von Komponenten anhand ihres Preises. Das Attribut proportionality in Zeile 1 zeigt an, dass die Sortierung invers zur natürlichen Ordnung erfolgen soll, sodass am Ende die günstigsten Komponenten den besten Rankingwert erhalten. Durch den SELECT-Ausdruck wird, wie in Codebeispiel 6.1, das Metadatum adressiert, dessen Wert für die Normalisierung genutzt werden soll (Variable v in Zeile 4). Liegen die Preise der Komponenten beispielsweise zwischen 0.49 und 2.99, so ergibt die Anwendung der Regel für eine Komponente mit einem Preis von 0.99 den Rankingwert 0.80. Ein höherer Preis (2.29) führt zu einer schlechteren Bewertung (0.28). Auch hier können die resultierenden Individuen von Metadata jeweils einem Kandidaten zugeordnet werden.

```
1 <rankingRule xsi:type="ccm:NormalizeRule" id="sort_by_price" proportionality="inverse"
2     metadata="SELECT ?metadata ?v
3         WHERE { ?metadata nfp:hasPricing ?p .
4             {?p nfp:hasValue ?v.} UNION {?p nfp:hasUpperBound ?v.} }">
5 </rankingRule>
```

Lst. 6.2: Beispiel einer Normalize Rule

Zur Abbildung nicht-numerischer Metadatenwerte dienen *Mapping Rules*, welche Komponenten je nach deren Metadatenbelegung vordefinierte Rankingwerte zuweisen. Codebeispiel 6.3 zeigt dies am Beispiel von Lizenzen, welche exemplarisch die Konzepte OSL, AFL, LGPL und GPL umfassen. In Zeile 1 wird durch das Attribut defaultrating ein Standardwert definiert, falls keines der Konzepte Anwendung findet. Das WHERE-Konstrukt adressiert das gesuchte Metadatum license, für welches die mapping-Elemente (Zeile 4–7) die Optionen samt Rankingwert definieren. Eine Komponente, die über eine GNU Lesser General Public License (LGPL) verfügt, erhält danach das doppelte Gewicht einer Komponente mit Open Software License (OSL).

```
<rankingRule xsi:type="ccm:MappingRule" id="open_license" defaultrating="0.2"
    metadata="SELECT ?metadata ?license
        WHERE { ?metadata nfp:hasLicense ?license. }">
    <mapping value="http://inf.tu-dresden.de/cruise/nfp.owl#GPL" rating="0.8"/>
    <mapping value="http://inf.tu-dresden.de/cruise/nfp.owl#LGPL" rating="0.6"/>
    <mapping value="http://inf.tu-dresden.de/cruise/nfp.owl#AFL" rating="0.4"/>
    <mapping value="http://inf.tu-dresden.de/cruise/nfp.owl#OSL" rating="0.3"/>
</rankingRule>
```

Lst. 6.3: Beispiel einer Mapping Rule

Wie bereits angedeutet, können alle Regeln als Teil des Kompositionsmodells instanziiert, oder von da aus in ihrer serialisierten Form referenziert werden. Für die darin verwendeten Ausprägungen der SPARQL-Anfragen gibt es verschiedene Implementierungsalternativen, auf die im Rahmen der Umsetzung in Abschnitt 7.2.1 eingegangen wird. Bereits an dieser Stelle sei angemerkt, dass die gezeigten Serialisierungsformen die einfachsten und performantesten Varianten darstellen, die im Falle der manuellen Definition durch Entwickler vorzuziehen sind.

6.1.3.2 Bewertung und Sortierung

Sind Sortierungsregeln vorhanden, die entweder global oder der betrachteten Vorlage zugewiesen sind, so werden sie zunächst auf alle Komponenten der Kandidatenmenge angewendet, wie im obigen Abschnitt beschrieben. Die Anwendung aller Regeln resultiert in einer Bewertung $rank_{component} \in [0, 1]$ für jede Komponente. Sie ergibt sich aus der Aggregation aller Einzelbewertungen. Die dafür eingesetzte Strategie ist konzeptionell freigestellt – im Normalfall wird der Mittelwert gebildet. Sind die Regeln unterschiedlich gewichtet (vgl. Abschnitt 5.2.1), werden die Gewichte bei der Mittlung berücksichtigt.
Abschließend wird die Bewertung $rank_{component}$ hinsichtlich des nicht-funktionalen Rankings mit dem der funktionalen Übereinstimmung $match_{component}$ aus der Matching-Phase verrechnet. Dazu wird letztere ebenfalls auf den Wertebereich $[0, 1]$ normiert und der Mittelwert $rating_{component}$ gebildet. Dieser bestimmt die Rangfolge innerhalb der Kandidatenmenge. Falls keine Ranking-Regeln anzuwenden sind, entspricht die Bewertung einer Komponente dem normierten $match_{component}$.
Die Ausgabe der Rangfolgebildung ist somit eine geordnete Liste mit der geforderten Anzahl funktional und nicht-funktional kompatibler Komponenten, denen jeweils eine Bewertung $rating_{component} \in [0, 1]$ zugewiesen ist. Die Liste wird an die Laufzeitumgebung zurückgegeben, die, wie im nächsten Abschnitt beschrieben, die Auswahl und Integration eines Kandidaten in die Anwendung vornimmt.

6.1.4 Auswahl und Integration

Im Anschluss an die Discovery erfolgt die Integration der „passendsten“ Komponente. Grundlage hierfür bietet die vom Komponenten-Repository zurückgelieferte Menge kompatibler Komponenten samt Bewertungszahl. Die Laufzeitumgebung kann zur Auswahl mehrere Strategien verfolgen – üblicherweise wird jedoch die Komponente mit dem höchsten Wert integriert werden. Alternativ kann dem Nutzer die Auswahl überlassen werden. Auf die alternativen Komponenten kann beim dynamischen Austausch zurückgegriffen werden, der automatisch bzw. durch Adaptionsaspekte,

oder aber durch den Nutzer initiiert wird. Zu diesem Zweck wurde ein Prototyp entwickelt, der Nutzern über ein Menü alternative Komponenten zu einer bestehenden, integrierten Instanz zum Austausch anbietet (vgl. Abschnitt 7.3.3).

Für die ausgewählte Komponente – in der Antwort vom CoRe lediglich über eine ID repräsentiert – fordert die Laufzeitumgebung deren Beschreibung (SMCDL, Abschnitt 5.1.3) an, die alle zum Laden und zur ihrer Initialisierung nötigen Informationen beinhaltet. Weiterhin wird das in Abschnitt 6.1.2.2 angesprochene *MatchingResult* geladen, welches als Ergebnis der Matching-Phase zur Verfügung steht. Wie oben beschrieben, beinhaltet es die Zuordnungsvorschriften für Properties, Operations, Events und Parametern zwischen Vorlage und zu integrierender Komponente. In der Praxis wird es vom CoRe im Sitzungskontext gehalten, die den Zusammenhang zwischen Discovery und Integration herstellt.

Die Komponentenbeschreibung wird danach von der Laufzeitumgebung interpretiert und die Komponente integriert. Einzelheiten der Integration werden in Abschnitt 6.2.3.2 behandelt. Der Integrationsprozess wird für alle Vorlagen (und in vereinfachter Form auch für alle konkreten Komponenten) des Kompositionsmodells durchlaufen. Parallel erfolgt die Interpretation der weiteren Teilmodelle, z. B. hinsichtlich der Etablierung von Kommunikationskanälen und dem Setzen des Layouts. Mit der erfolgreichen Integration der letzten Komponente ist die Anwendung nutzbar. Der nächste Abschnitt beschreibt die der Ausführung zugrunde liegende Laufzeitumgebung und geht dabei auch auf die bereits angesprochenen Aspekte des Komponenten-Lebenszyklus und der Mediation ein.

6.2 Kompositionsinfrastruktur und Laufzeitumgebung

An dem im letzten Abschnitt vorgestellten Integrationsprozess sind eine Reihe von Modulen beteiligt, deren Zusammenhang und Verortung innerhalb der Kompositionsinfrastruktur in Abbildung 4.4 verdeutlicht wurden. Die folgenden Abschnitte sind entsprechend der darin angedeuteten Aufteilung in die Verwaltungs- und Kompositionsbelange gegliedert.

Zunächst wird auf die Dienste zur Verwaltung von Komponenten und Domänenmodellen eingegangen. Im Anschluss steht die Laufzeitumgebung (MRE) im Mittelpunkt der Betrachtung. Nach einem architektonischen Überblick werden der Lebenszyklus von Mashup-Komponenten sowie deren dynamische Integration vorgestellt. Die Erläuterung der Koordinations- und Mediationskonzepte rundet das Teilkapitel ab.

6.2.1 Verwaltung von Komponenten und Domänenwissen

Grundlage für die dynamische Auswahl und Integration von Komponenten ist die Verfügbarkeit möglichst vieler, qualitativ hochwertiger Komponenten. In Anlehnung an die Trennung von *Service Consumer* und *Service Registry* in SOA, empfiehlt sich auch für Mashup-Komponenten die Verwaltung durch eine unabhängige Instanz. Daneben müssen auch die zur semantischen Typisierung genutzten Domänenmodelle verwaltet werden, da sie zur Laufzeit für die Überbrückung etwaiger syntaktischer Differenzen benötigt werden.

Im Folgenden werden die entsprechenden Verwaltungsmodule der Kompositionsinfrastruktur kurz hinsichtlich ihrer Rolle und Aufgaben im Zusammenspiel mit der dynamischen Integration und Ausführung kompositer Anwendungen vorgestellt.

Komponentenverwaltung

Die Trennung von Kompositionsplattform und Komponentenverwaltung ist mit einigen Vorteilen verbunden. So kann es u. a. mehrere Repositories geben, aus denen eine Laufzeitumgebung Komponenten bezieht, und letztere können anwendungs- und plattformübergreifend zum Einsatz kommen. Die regelmäßige Überprüfung von Validität und Konsistenz der Beschreibungen seitens der verwaltenden Instanzen kann zudem Fehlern bei der Integration vorbeugen und somit die Performanz und Qualität der dynamisch komponierten Lösungen steigern.

Im vorliegenden Konzept übernimmt das Component Repository (CoRe) die Aufgabe der Verwaltung von Komponentenbeschreibungen. Es bietet dazu eine CRUD-Schnittstelle an, um die Registrierung und Aktualisierung von Komponenten über die in Abschnitt 5.1.3 vorgestellten Deskriptoren, den Abruf von Beschreibungen sowie deren Entfernung zu ermöglichen. Im Falle der Registrierung von MCDL- oder SMCDL-Beschreibungen erfolgt intern die automatische Abbildung der Elemente aus XML auf Konzepte der MCDO. Letztere bildet die Grundlage der Verwaltung und Persistierung im CoRe. Zusätzlich werden die enthaltenen funktionalen und nichtfunktionalen Referenzen jeweils als Konzepte der adressierten Modelle instanziiert.

Auf Basis des semantischen Modells ermöglicht das CoRe schließlich den Abruf von Deskriptoren im jeweils gewünschten Format (MCDL, SMCDL, MCDO). Die Serialisierung kann dabei – je nach Anforderungen der Laufzeitumgebungen – variiert werden und neben XML beispielsweise in JSON erfolgen.

Den einfachsten Anwendungsfall bildet die Anfrage nach einer Komponente über ihre ID. Es besteht jedoch auch die Möglichkeit, deutlich komplexere Anfragen an das Modell mit Hilfe dedizierter Methoden oder SPARQL zu stellen. Dadurch ist die Filterung nach bestimmten Merkmalen der Ausführungsplattform oder nach Stichwörtern möglich, um die Suche im Autorenprozess zu unterstützen.

Discovery

Da der in Abschnitt 6.1.4 beschriebene Discovery-Prozess auf dem semantischen Modell des CoRe stattfindet, empfiehlt sich mit Blick auf die Performanz die direkte Integration der entsprechenden Algorithmen. Die Trennung von Komponentenverwaltung und Discovery ist prinzipiell möglich, würde jedoch einen umfangreichen Austausch der jeweils benötigten Modelldaten sowie ggf. die Synchronisation der Modelle auf beiden Seiten nach sich ziehen (vgl. Verteilungsalternativen, Abschnitt 7.2.1).

Folglich kann das CoRe die in Abschnitt 6.1.1 vorgestellte Anfrage zur Suche von Komponenten auf Basis semantischer Abstraktionen verarbeiten. Ausgehend von der semantischen Zielbeschreibung in Form eines Templates liefert es eine gewichtete Liste passender Kandidaten, die durch die anfragende MRE ausgewertet werden kann. Der berechnete Deckungsgrad sowie die Zuordnungsregeln werden im Sitzungskontext vorgehalten und können für die zu integrierende Komponente später abgerufen werden.

Typ- und Modellverwaltung

Der im ersten Teilkapitel vorgestellte Integrationsprozess machte deutlich, dass die Abstraktionskonzepte im Kompositionsmodell in Verbindung mit der semantischen Typisierung die Integration partiell passender Komponenten zur Laufzeit ermöglichen. Diese gesteigerte Flexibilität und Interoperabilität macht es erforderlich, zwischen semantisch kompatiblen, jedoch syntaktisch verschiedenen Entitäten zu vermitteln. Grundlage für diese Vermittlung, die in Abschnitt 6.2.4.2 ausführlich beschrieben wird, bilden die Referenzen der Komponentenschnittstellen auf funktionale und nicht-funktionale semantische Konzepte.
Die Verwaltung der entsprechenden Domänenmodelle übernimmt in der konzeptionellen Architektur das Semantic Type Repository (TyRe). Neben den Modellinstanzen verwaltet es die Liftings und Lowerings der Konzepte, auf die im Rahmen der semantischen Annotation in Abschnitt 5.1.3.3 eingegangen wurde. Sie beschreiben die Abbildung zwischen syntaktischer und semantischer Repräsentation eines Datums in beide Richtungen und werden zur Mediation über die semantische Ebene benötigt (Abschnitt 6.2.4.2). Bei der Nutzung XML-basierter Datentypen kommen dafür i. d. R. XSLT-Skripte zum Einsatz.
Für den Zugriff auf diese Informationen bietet das TyRe eine Schnittstelle zur Registrierung und Aktualisierung, den Abruf und die Entfernung von Konzepten, entsprechenden Typdefinitionen, Liftings und Lowerings. Die Adressierung erfolgt über die URI des betreffenden Ontologiekonzeptes. In der verteilten Kompositionsinfrastruktur kann die Typverwaltung strukturell mit dem CoRe zusammenfallen und ggf. auch auf Modellebene mit der Komponentenverwaltung integriert werden. Verschiedene Verteilungsmodelle werden in Abschnitt 7.2.1 diskutiert.

6.2.2 Aufbau der Laufzeitumgebung (MRE)

Der zentrale Bestandteil der Referenzarchitektur zur Ausführung kompositer interaktiver Webanwendungen ist die Laufzeitumgebung – die Mashup Runtime Environment (MRE) aus Abbildung 4.4. Der konzeptionelle Aufbau dieses Systems wird in Abbildung 6.5 anhand der wichtigsten Module veranschaulicht, unabhängig von deren technologischen Umsetzung oder Client-Server-Verteilung.
Die MRE dient in erster Linie der Ausführung und Verwaltung der *Composite Application* (oben), d. h. der modellierten Anwendung und ihrer Komponenten. Da letztere alle dem gleichen Komponentenmodell unterliegen, können sie nach den gleichen Prinzipien integriert, verknüpft und angesprochen werden. Darunter sind die Bestandteile der Laufzeitumgebung zu sehen, deren Verantwortlichkeiten im Folgenden näher erläutert werden. Sie stützen sich wiederum auf externe Dienste ab. Dazu zählen das CoRe, welches der Verwaltung von Komponenten dient, Ressourcen, die zur Ausführung der Komponenten geladen werden müssen sowie weitere Backend-Dienste, die durch Komponenten angebunden werden, um Daten und Anwendungslogik bereitzustellen. Auf der rechten Seite ist das Adaptionssystem zu erkennen, welches mit der MRE gekoppelt ist und seinerseits den *Context Service* anbindet, dem die Verwaltung der konsistenten Kontext-Wissensbasis obliegt. Je nach Verteilungsgrad der Laufzeitumgebung können auch Module der MRE in separate Dienste ausgelagert werden. Dies wird in Abschnitt 7.2.2.1 am Beispiel der Mediation diskutiert.

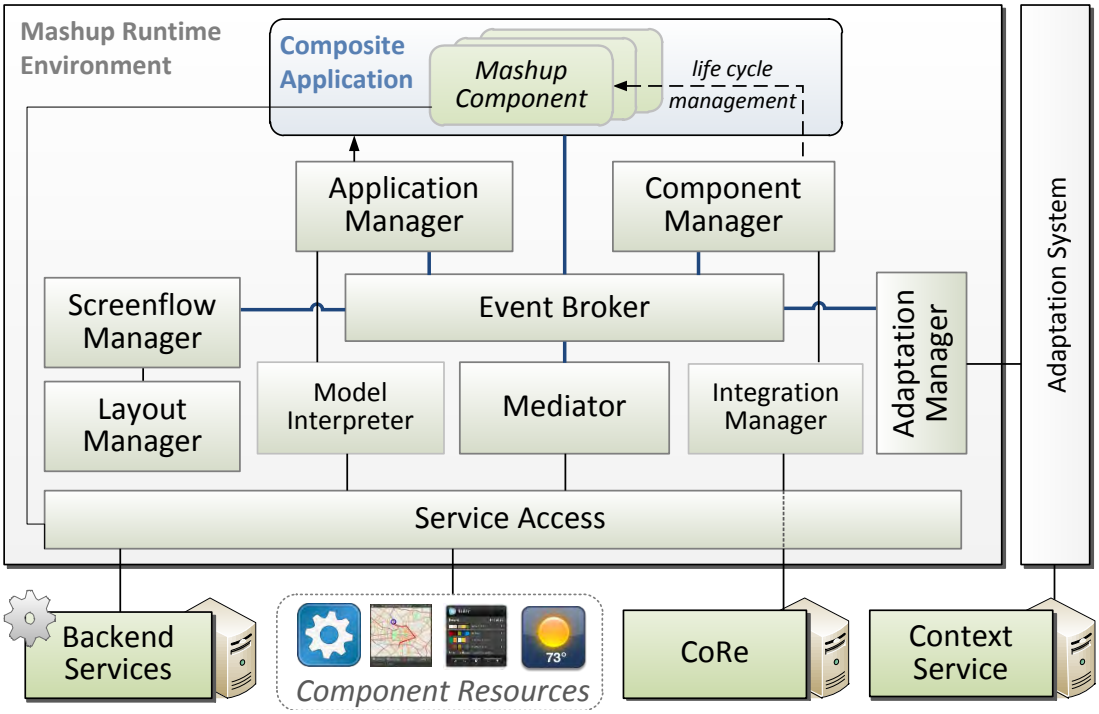

Abb. 6.5: Konzeptioneller Überblick über das Mashup Runtime Environment

Die Arbeit der MRE beginnt mit einer Client-Anfrage nach einer kompositen Anwendung, die in Form eines Kompositionsmodells oder durch vorgelagerte Transformationen bereits in ausführbarem Code vorliegt. Die folgenden Module sind an der Integration und Ausführung dieser Anwendung beteiligt:

Application Manager

Dieses Modul ist für die Initialisierung aller anderen MRE-Module sowie die Verwaltung der Gesamtanwendung zuständig. Grundlage hierfür bieten der aus dem Kompositionsmodell generierte Quellcode oder die Information, die durch den **Model Interpreter** bereitgestellt werden. Zusätzlich verwaltet und sichert der Application Manager den Zustand der Anwendung persistent und deckt die globale Fehlerbehandlung ab. Zur Integration und Verwaltung von Komponenten greift er auf die Funktionalitäten des *Component Managers* zu.

Component Manager

Die Verwaltung des Lebenszyklus' von Komponenten (Abschnitt 6.2.3.1) von ihrer Integration über die Instanziierung, Initialisierung und Ausführung bis hin zur Zerstörung übernimmt dieses Modul. Da alle Mashup-Komponenten dem gleichen Komponentenmodell genügen, kann ihre Verwaltung einheitlich – mit einigen Erweiterungen für Bestandteile der Präsentationsschicht, z. B. hinsichtlich ihrer Anordnung auf der Oberfläche – erfolgen. Vor der Instanziierung wertet der *Component Manager* die aus dem Modell verfügbaren Komponenteninformationen aus und reicht sie zur dynamischen Integration an den *Integration Manager* weiter.

Integration Manager

Die dynamische Einbindung von Komponenten wird durch dieses Modul ausgeführt. Dazu erfolgt der Abruf von Komponentendeskriptoren bei CoRe mit Hilfe konkreter

Referenzen oder semantischer Zielbeschreibungen (*Templates*), wie in Abschnitt 6.1.1 dargelegt. Der Integration liegen die Informationen der zurückgelieferten Bindings zugrunde, die Aufschluss über Abhängigkeiten und Zugriffsmethoden bieten. Nach dem Abschluss der Einbindung wird der Component Manager informiert, um mit der Initialisierung fortzufahren.

Screenflow Manager

Die Interpretation der im Modell definierten Sichten übernimmt der Screenflow Manager. Er registriert und verarbeitet auslösende Events für den Wechsel zwischen Ansichten (*Views*) und setzt diese um, indem er die enthaltenen Layouts durch den Layout Manager zeichnen lässt und die Sichtbarkeit der enthaltenen Komponenten je nach Bedarf ändert.

Layout Manager

Dieses Modul wertet die Informationen des *Layout Models* aus und wendet sie auf die Komponenten der Präsentationsebene an. Dazu setzt es die entsprechenden Layouts in der Technologie der Plattform um, konfiguriert die enthaltenen Komponenten, z. B. hinsichtlich ihrer Größe, und ordnet sie wie modelliert auf der Oberfläche an. Der Aufruf zum Rendering eines Layouts erfolgt durch den o. g. Screenflow Manager.

Event Broker

Dem Komponentenmodell folgend, läuft die Kommunikation innerhalb einer Anwendung ereignisorientiert ab. Die MRE stellt dazu den *Event Broker* bereit, welcher die getypten Links des Kompositionsmodells auf eine Publish/Subscribe-Architektur abbildet, an die die jeweiligen Ereignisse und Operationen angebunden sind. Auch die Kommunikation mit und zwischen den Modulen der MRE erfolgt z. T. asynchron über den Broker. Neben der Vermittlung von Nachrichten und der Überprüfung ihrer syntaktischen Korrektheit erfolgt „auf“ den Links zudem die semantische Mediation (Abschnitt 6.2.4.2) durch den *Mediator*.

Mediator

Zur Steigerung der Interoperabilität zwischen Komponenten können – wie in Abschnitt 6.1.2 bereits angeklungen ist – syntaktische Differenzen zwischen Komponentenschnittstellen auf semantischer Ebene überbrückt werden. Diese Aufgabe übernimmt der *Mediator*, der mit Hilfe der im letzten Abschnitt angesprochenen Liftings und Lowerings die Transformation der Nutzdaten auf der semantischen Ebene vornimmt.

Adaptation Manager

Die dynamische Rekonfiguration, der Austausch von Komponenten sowie die komponentenübergreifende Adaption der Anwendung, z. B. von Layout und Kommunikationskanälen, in Abhängigkeit vom Nutzer und seinem Kontext (Endgerät, Präferenzen, Umgebung, etc.) übernimmt dieses Modul. Es stellt die Schnittstelle zum Adaptionssystem dar, welches in Abschnitt 6.3 ausführlich erläutert wird und zur Kontextverwaltung auf einen dedizierten Kontextdienst zugreift.

Service Access

Für den Zugriff auf funktionale und Datendienste bietet die MRE die *Service-Access*-Schnittstelle. Diese erlaubt Komponentenentwicklern auf der einen Seite den

einheitlichen Zugriff auf verschiedenste Web-Dienste und Ressourcen im Backend. Die MRE wird auf der anderen Seite in die Lage versetzt, Anfragen zu prüfen, zu authentifizieren und ggf. über Proxies umzuleiten, was die Nutzung in gesicherten Netzwerken sowie – im Fall einer clientseitigen Ausführung – die Umgehung der *Same Origin Policy* [@Zalewski, 2011] ermöglicht.

In ihrer Gesamtheit decken die Module der MRE alle Anforderungen aus Abschnitt 2.3 ab: Durch die direkte Interpretation des Kompositionsmodells wird die modellbasierte Entwicklung von Mashup-Anwendungen unterstützt. Sie mündet in der dynamischen, kontextabhängigen Integration von Komponenten zur Laufzeit auf Basis der Abstraktionen des Modells. Diese sind lose über den Event Broker gekoppelt, wobei durch Mediationsmechanismen auf der Schnittstellen- und Datenebene die Interoperabilität auch bei syntaktischen Differenzen sichergestellt wird. Eine einheitliche Dienstzugriffsschicht, die von heterogenen Diensttypen und Protokollen abstrahiert, erlaubt domänenübergreifende Dienstaufrufe und aus Sicht der Plattform die Validierung und Beeinflussung der Kommunikation im Backend. Durch die nahtlose Integration des Adaptionssystems und die Anbindung des Kontextdienstes können schließlich Anpassungen auf Komponenten- und Kompositionsebene zur Laufzeit durchgeführt werden, wie sie durch die Aspekte im Kompositionsmodell vorgeschrieben wurden.
Die folgenden Abschnitte gehen auf ausgewählte Aspekte der Ausführung und das Zusammenspiel der Module genauer ein. Zunächst wird die dynamische Integration und Verwaltung von Mashup-Komponenten anhand deren Lebenszyklus beleuchtet. Danach gibt Abschnitt 6.2.4 einen Einblick in die Funktionsweise der losen Kopplung von Komponenten und stellt die damit verbundenen Mediationskonzepte vor.

6.2.3 Dynamische Integration und Verwaltung von Komponenten

Die dynamische Integration von Mashup-Komponenten und ihre Verwaltung führen gegenüber statisch komponierten Anwendungen zu erhöhten Anforderungen. Die folgenden Abschnitte geben Aufschluss über den Lebenszyklus der Komponenten und stellen den Ablauf bei der dynamischen Integration genauer dar.

6.2.3.1 Phasen im Komponenten-Lebenszyklus

Komponenten gemäß dem vorgestellten universellen Komponentenmodell unterliegen zur Laufzeit einem vordefinierten Lebenszyklus. Dieser wird durch den Component Manager geregelt und über *Life-Cycle-Events* publiziert, sodass beliebige Module, z. B. mit dem Ziel der Adaption, darauf reagieren können. Der Zustandswechsel erfolgt über *Life-Cycle-Methoden*, die von Komponentenentwicklern unterstützt werden müssen und in der folgenden Erläuterung der einzelnen Phasen jeweils Erwähnung finden. Abbildung 6.6 illustriert die verschiedenen Phasen in ihrer Gesamtheit.

Registered Zum Zeitpunkt der Initialisierung einer Mashup-Anwendung existieren zu integrierende Komponenten als unabhängige Implementierungen, bestehend aus potentiell mehreren Ressourcen und durch eine der in Abschnitt 5.1.3 vorgestellten Sprachen beschrieben. Über diese Beschreibung sind sie in einem oder mehreren Repositories registriert.

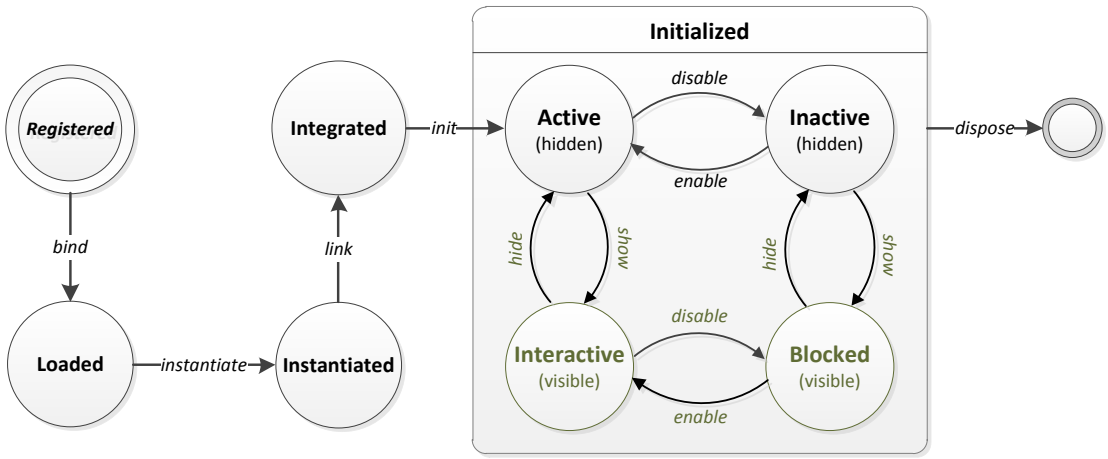

Abb. 6.6: Lebenszyklus einer Mashup-Komponente

Loaded Nach der Ermittlung einer passenden Komponente folgt ihre Bindung durch die MRE. Dazu werden zunächst all ihre Abhängigkeiten, wie benötigte Programm-Bibliotheken, CSS-Definitionen, Bilder, usw. geladen.

Instantiated Sind alle nötigen Bestandteile erfolgreich geladen, wird die Komponente instanziiert. Dazu erfolgt der Aufruf ihres Konstruktors und der Component Manager übernimmt die Verwaltung der Instanz.

Integrated Eine erfolgreich instanziierte Komponente kann nun in die Anwendung integriert werden. Dies umfasst ihre Konfiguration gemäß dem Kompositionsmodell sowie ihre Einbindung in den Daten- und Kontrollfluss der Anwendung. Eingehende Nachrichten werden zunächst vorgehalten und erst an die Komponente weitergereicht, wenn diese erfolgreich initialisiert wurde.

Initialized Bei der Initialisierung erhält die Komponente Zugriff auf verschiedene Objekte der MRE, auf die im nächsten Abschnitt genauer eingegangen wird. Zugriff und Nutzung dieser Objekte sind durch die plattformunabhängige Runtime-API festgelegt, d. h. Struktur und Verhalten sind unabhängig von der eingesetzten Technologie (Java, JavaScript, Flash, etc.). Zudem kann die Initialisierung durch Komponenten genutzt werden, um interne abhängige Variablen zu setzen.

Eine initialisierte Komponente kann zur Laufzeit entweder aktiv oder inaktiv, und im Fall von UI-Komponenten zusätzlich sichtbar oder unsichtbar sein. Der Wechsel zwischen diesen Zuständen erfolgt durch den Aufruf der entsprechenden Life-Cycle-Methoden. Es ergeben sich die folgenden Zustände:

Active Nach ihrer Initialisierung ist eine Komponente zunächst aktiv, d. h. sie stellt die erwartete Funktionalität über ihre öffentliche Schnittstelle zur Verfügung. Ist sie nicht aktiv, kann eine Komponente über ihre Methode enable aktiviert werden.

Inactive Durch den Aufruf der Methode disable kann eine Komponente deaktiviert werden. Dies geschieht immer dann, wenn sie nicht Teil der aktiven Sicht ist bzw. von der Kommunikation ausgeschlossen ist (vgl. Screenflow Model, Abschnitt 5.2.4). Die Behandlung dieses Zustands muss durch die

Komponentenentwickler erfolgen, z. B. indem Aktualisierungen durch den Aufruf externer Dienste zunächst ausgesetzt werden.

Interactive Eine aktive Komponente wird durch den Aufruf ihrer Methode show für den Nutzer sichtbar bzw. *gerendert*. Die nötigen Informationen, z. B. die Referenz des Containerobjekts, in welches sich die Komponente zeichnet, werden ihr bei der Initialisierung übergeben. Die Methode hide löst die Transition zurück in den Zustand *Active* aus.

Blocked Wird eine sichtbare Komponente inaktiv, so ist sie für den Nutzer blockiert – er kann nicht mehr mit ihr interagieren. Insbesondere für die Abbildung sequentieller Abläufe in der Benutzeroberfläche, z. B. bei formularbasierten Anwendungen, ist dieser Zustand nützlich, um Fehleingaben zu verhindern und korrekte Abläufe sicherzustellen. Das Black-Box-Prinzip macht es allerdings nötig, dass die modale Sperre explizit durch Komponentenentwickler umgesetzt wird, da die MRE keinerlei Kontrolle und Wissen über die Interna und UI-Elemente der Komponenten hat.

Disposed Bei der Entfernung einer Komponente, z. B. im Rahmen eines Komponentenaustauschs oder in Folge einer entsprechenden Adaption, wird deren Methode dispose aufgerufen. Diese gibt per Definition alle intern generierten Ressourcen frei. Gleichzeitig scheidet die Komponente aus allen Kommunikationsbeziehungen aus, bevor ihre Instanz gelöscht wird. Der Ablauf des Komponententauschs wird genauer in Abschnitt 6.3.3 erläutert.

Die beschriebenen Phasen machen deutlich, dass Komponentenentwicklern eine große Verantwortung bei der korrekten Umsetzung der Life-Cycle-Methoden zukommt. Da es sich bei ihnen um professionelle Programmierer handelt und die Konformität hinsichtlich der geforderten API werkzeuggestützt überprüft werden kann, können sie dieser Verantwortung gerecht werden.
Der folgende Abschnitt stellt die dynamische Integration von Mashup-Komponenten in den Mittelpunkt und verdeutlicht deren Ablauf anhand der genannten Phasen.

6.2.3.2 Ablauf der dynamischen Integration

An die Modellinterpretation und Suche nach Komponenten entsprechend der Discovery-Verfahren aus dem ersten Teilkapitel schließt sich deren Integration in die komposite Anwendung an. Wie in Abschnitt 6.1.4 skizziert, bilden die Komponentenbeschreibung in SMCDL sowie das in Abschnitt 6.1.2.2 vorgestellte *MatchingResult* dafür den Ausgangspunkt. Beide werden durch den *Integration Manager* vom CoRe abgerufen.
Alle zum **Laden** einer Komponente nötigen Informationen sind im Binding-Teil ihrer Beschreibung enthalten. Zunächst werden durch den Integration Manager alle Ressourcen abgerufen, die darin als *dependencies* deklariert sind. Andere Abhängigkeiten, z. B. zwischen den Komponenten, existieren per Definition nicht. Aus Gründen der Robustheit und Sicherheit muss für die Isolation der Komponenten gesorgt werden. Deshalb empfiehlt sich das Laden und die spätere Initialisierung innerhalb von *Sandboxes*, wie von Van Acker et al. (2011) vorgeschlagen.
Nutzen mehrere Komponenten die gleichen Bibliotheken, kann deren mehrfacher Abruf entfallen, was eine effiziente Verwaltung und Redundanzchecks nötig macht.

Der Vorgang bedarf ebenfalls Mechanismen zur Fehlerbehandlung, falls nicht alle Ressourcen geladen werden können. Da die Verfügbarkeit aller Abhängigkeiten durch das CoRe geprüft werden sollte, ist davon auszugehen, dass es sich um ein temporäres Problem handelt. Schlägt der Ladeversuch jedoch wiederholt fehl, muss eine alternative Komponente gewählt werden.
Wenn alle benötigten Ressourcen geladen werden konnten, befindet sich die Komponente im Zustand *Loaded*, wie im letzten Abschnitt beschrieben.

Vor der **Instanziierung** wird für jede Komponente ein Stellvertreter bzw. *Wrapper* angelegt. Dieser weist die öffentliche Schnittstelle gemäß der Vorlage aus dem Kompositionsmodell auf und nimmt alle nötigen Abbildungen auf interne Schnittstellen und Datenmodelle vor. Die Bindung der eigentlichen Komponente (und später auch der Komponententausch) erfolgt für die Anwendung verdeckt „hinter“ diesem Stellvertreter. Grundlage für dessen Konfiguration bilden das *MatchingResult*, welches etwaige Schnittstellenabweichungen zwischen der Vorlage im Modell und der Komponentenschnittstelle beschreibt, sowie die Zugriffsinformationen aus dem Komponenten-Binding. Eine genauere Erläuterung der Abbildungs- und Mediationskonzepte erfolgt in Abschnitt 6.2.4.2.
Zur Instanziierung wird schließlich auf den Konstruktor-Code zurückgegriffen, der als obligatorischer Teil des MCDL-*Bindings* definiert ist. Verläuft sie erfolgreich, befindet sich die Komponente im Zustand *Instantiated* und der Component Manager übernimmt die Verwaltung der Instanz.
Probleme bei der Instanziierung können durch Übertragungsfehler beim Laden der Ressourcen hervorgerufen werden, oder durch Komponenten selbst. Vor der Wahl einer alternativen Komponente wird deshalb ein zweiter Initialisierungsversuch nach dem erneuten Laden der Abhängigkeiten vorgenommen.

An die erfolgreiche Initialisierung schließt sich die **Integration** der Komponente an, die zwei Teilschritte umfasst. Zunächst erfolgt die anwendungsspezifische Konfiguration, d. h. die Properties der Komponente werden entsprechend der Belegungen im Kompositionsmodell gesetzt. Danach wird die Komponente in den Kontroll- und Datenfluss der Anwendung eingebunden. Dazu interpretiert der Event Broker das Kommunikationsmodell (Abschnitt 5.2.2) und registriert ihre Events und Operations an den entsprechenden Links. Bereits in dieser Phase erfolgt jeglicher Zugriff auf die Komponente über deren Stellvertreter, sodass Konfiguration und Verknüpfung rein auf Basis der Modellinformationen, unabhängig von Abweichungen der ausgewählten Implementierung erfolgen können.
Auch bei der Integration können Fehler auftreten. Sie deuten darauf hin, dass die Schnittstelle der integrierten Komponente nicht der über die MCDL deklarierten entspricht, oder dass die Zuordnungsregeln aus der Discovery-Phase fehlerhaft sind. Letztere können neu angefordert werden – erneute Fehler führen aber wiederum zum Austausch.

Zuletzt kommt es zur **Initialisierung** der Komponente. Sie erfolgt durch den Aufruf der Life-Cycle-Methode `init`. Über deren einzigen Aufrufparameter – den `ComponentContext` – erhält die Komponente Zugriff auf verschiedene Objekte der Laufzeitumgebung. Für UI-Komponenten stehen beispielsweise die Stylinginformation aus dem Kompositionsmodell sowie eine Referenz auf das Objekt zur Verfügung, in das die Komponente gezeichnet werden soll. Eine wichtige Rolle nimmt

der *EventHandler* ein – er bietet der Komponente die Möglichkeit, Ereignisse zu publizieren. Allerdings handelt es sich dabei nicht um den Event Broker, sondern um den o. g. Wrapper, der die publizierten Informationen so ggf. noch transformieren kann. Für den Dienstzugriff wird der Komponente schließlich eine Referenz auf die Service-Access-Schnittstelle angeboten. All jene Objekte der MRE folgen einer generischen Spezifikation, allerdings können für sie je nach Technologie der Plattform und integrierten Komponente verschiedene Umsetzungen existieren.
Fehler bei der Initialisierung können von Komponenten durch Fehler-Ereignisse angezeigt und durch die MRE behandelt werden. Im Normalfall kommt es zu einem erneuten Initialisierungsversuch. Falls dieser fehlschlägt, oder anderweitige Probleme, z. B. *Exceptions*, auftreten, muss die Komponente entfernt werden, was u. a. die Deregistierung beim Event Broker nach sich zieht.

Es ist ersichtlich, dass Fehlern bei der Integration durch umfassende Validierung und Konsistenzsicherung bei der Komponentenverwaltung vorgebeugt werden kann. Die Verfügbarkeit aller Ressourcen und die syntaktische Korrektheit der angegebenen Code-Teile, z. B. von Konstruktor und Destruktor, können bereits vor der Auslieferung an eine MRE geprüft werden. Es ist ebenfalls möglich, die Instanziierung und Initialisierung auf Seiten des CoRe zu testen, allerdings bedarf es dazu plattformspezifischer Emulatoren. Entsprechende Qualitätssicherungsverfahren werden empfohlen, gehen jedoch über die Schwerpunkte der Arbeit hinaus.
Die erfolgreiche Integration einzelner Mashup-Komponenten ist allerdings nur der erste Schritt zu einem voll funktionsfähigen Mashup. Gegenüber Dashboard-Anwendungen besteht ein Mehrwert in der Kommunikation und Koordination zwischen Komponenten, auf die im nächsten Abschnitt genauer eingegangen wird.

6.2.4 Kommunikationsinfrastruktur und Mediation

Die Umsetzung der Kommunikation und Koordination zwischen Komponenten stellt aufgrund der unabhängigen Entwicklung ihrer Bestandteile eine Herausforderung dar. Dieser Abschnitt widmet sich den dafür nötigen Kommunikations- und Mediationsmechanismen der MRE, welche den Daten- und Kontrollfluss unter Beachtung der losen Kopplung aller Bestandteile ermöglichen.
Zunächst wird der prinzipielle Ablauf der Kommunikation entsprechend den im Communication Model (Abschnitt 5.2.2) vorgestellten Mustern bzw. Links dargestellt. Danach widmet sich Abschnitt 6.2.4.2 den Interoperabilitätskonzepten auf Schnittstellen- und Datenebene.

6.2.4.1 Ablauf der Kommunikation

Die angestrebte lose Kopplung von Mashup-Komponenten übernimmt in der MRE der *Event Broker*. Er verwaltet alle Kommunikationskanäle einer kompositen Anwendung gemäß der Links im Kompositionsmodell und vermittelt darüber kommunizierte Nachrichten zwischen den Komponenten. Letztere bleiben vollständig unabhängig voneinander, zumal das Broker- bzw. Mediator-Muster [Gamma et al., 1995] die stärkste Entkopplung und eine höhere Flexibilität als vergleichbare Ansätze, wie das Bus-System, bietet [Eugster et al., 2003].

Nachrichtenobjekt

Der Austausch von Informationen erfolgt in Form eines Nachrichtenobjektes, welches die in der Schnittstellenbeschreibung spezifizierten Parameter sowie Steuerinformationen enthält. Letztere ermöglichen u. a. die Zuordnung von Nachrichten und die Fehlerbehandlung. Zur Trennung von Nutz- und Steuerdaten zerfällt jede Nachricht, wie von Hohpe und Woolf (2003) vorgeschlagen, in *Body* und *Header*.
Die Nutzung eines einheitlichen Nachrichtenobjektes ist mit einer Reihe von Vorteilen verbunden. So können die auszutauschenden Daten auf einer gemeinsamen formalen Grundlage und plattformunabhängig repräsentiert werden. Komponentenentwickler können eine einfache, leichtgewichtige API nutzen, um ihre Daten zu versenden, und durch den Einsatz entsprechender, vordefinierter Header-Informationen können Anwendungs-, Life-Cycle- und Fehler-Ereignisse einheitlich repräsentiert werden. Die Serialisierung der entstehenden Objekte kann – falls nötig – durch die MRE plattformspezifisch, z. B. in XML oder JSON, erfolgen.

Im Folgenden soll kurz auf die wichtigsten Informationen des Nachrichtenobjektes eingegangen werden, die für die späteren Konzepte zur Nachrichtenvermittlung und Mediation von Relevanz sind. Im Header zählen dazu:

Status Über den Erfolg oder Misserfolg einer Nachrichtenvermittlung gibt ein Statuscode gemäß der HTTP-Spezifikation [@Fielding et al., 1999] Auskunft. Statuscode 200 symbolisiert beispielsweise eine erfolgreiche Kommunikation, während Code 400 auf eine fehlerhafte Anfrage hinweist.

Name Die Zuordnung eines Ereignisses zu einem Link durch den Event Broker erfolgt über die Kombination aus dem hier angegebenen Namen des Ereignisses und der eindeutigen ID der Komponente, die jeder Nachricht automatisch durch den o. g. Wrapper hinzugefügt wird.

Serialization Die Serialisierungsform der übermittelten Parameter kann über dieses optionale Feld angegeben werden. Standardmäßig werden sie in Form einfacher oder komplexer XML-Datentypen serialisiert. Je nach Plattform sind jedoch andere Formen, wie z. B. JSON oder RDF, für die effiziente Verarbeitung geeigneter. Deshalb ist die Unterstützung alternativer Groundings möglich, deren Nutzung hier im Header angezeigt wird, und die in der MCDL angezeigt werden müssen.

CallbackID Für die Zuordnung zusammengehöriger Nachrichten, insbesondere bei der Verknüpfung mit BackLinks, wird diese ID benötigt. Durch sie kann der Event Broker Rückgabe-Ereignisse erkennen und an die jeweils passende Callback-Operation weiterleiten. Die CallbackID wird vom Broker dem auslösenden Event hinzugefügt und muss vom Callback-Event übernommen werden.

SyncThreshold Das im Kommunikationsmodell definierte Zeitintervall zur Begrenzung periodischer Aktualisierungen wird durch dieses Feld ausgedrückt. Ist es nicht belegt, handelt es sich um eine einfache Anfrage, der Wert „0“ impliziert permanente Aktualisierungen, und jeder sonstige numerische Wert zeigt die Anzahl der Sekunden an, die zwischen zwei Rückrufen minimal liegen müssen.

Der Body der Nachricht enthält die eigentliche Nutzdaten, d. h. die Menge der in der Komponentenbeschreibung definierten Parameter des Events. Sie liegen in Form von Name-Wert-Paaren vor.

Nachrichtenvermittlung

Die Vermittlung von Nachrichten erfolgt über den Event Broker. Jede Komponente kann eigene Ereignisse an dem Broker publizieren, wobei es sich um die beschriebenen Life-Cycle-Events, implizite Events zur Anzeige von Property-Änderungen, oder Events der öffentlichen Komponentenschnittstelle handeln kann.
Zugriff auf den Event Broker erhält eine Komponente über das *ComponentContext*-Objekt, welches ihr bei der Initialisierung übergeben wird (vgl. Abschnitt 6.2.3.1). Neben der Methode publish zur Publikation von Events bietet der Broker auch eine API zur Erstellung von Nachrichtenobjekten an. Somit kann die Konsistenz und Validität jeder erstellten Nachricht durch die MRE selbst sichergestellt werden.
Die Grundfunktionalität des Event Brokers als Modul der MRE umfasst die Verwaltung sämtlicher Kommunikationsbeziehungen der Komposition. Neben der Erstellung und Entfernung von Link-Kanälen müssen gleichsam alle angeschlossenen Kommunikationspartner sowie etwaige Abbildungsregeln auf Signaturebene verwaltet werden. Die folgenden Abschnitte geben Aufschluss über die Funktionsweise des Event Brokers zur Realisierung der modellierbaren Verknüpfungsformen.

Die Umsetzung der unidirektionalen Verknüpfung von Komponenten über **Links** stellt die einfachste Form der Nachrichtenvermittlung dar. Eine Komponente erstellt und konfiguriert dazu über den Event Broker ein Nachrichtenobjekt und übergibt es schließlich über die Methode *publish* an den Broker.
Durch die Kombination aus Komponenten- und Event-ID ist die Nachricht eindeutig bestimmt und kann den Links des Modells zugeordnet werden. Nach Prüfung der Validität der Nachricht und der Typisierung der enthaltenen Parameter wird sie an alle Operationen weitergeleitet, die an die gefundenen Links angeschlossen sind. Sollten auf Sender- oder Empfängerseite Parameter Mappings modelliert sein (vgl. Abschnitt 5.2.2.2), so werden diese vor bzw. nach der Vermittlung interpretiert, was die Umsortierung und Entfernung von Parameter zur Folge haben kann. Differenzen auf der Datenebene werden zudem durch Mediationsmechanismen aufgelöst, die den Schwerpunkt von Abschnitt 6.2.4.2 bilden.
Als Rückgabewert der Methode publish dient eine Statusnachricht, deren Statuscode Aufschluss über den Erfolg der Vermittlung gibt (s. o.). Ob und wie sie ausgewertet wird, bliebt den Komponentenentwicklern überlassen.

Die Umsetzung von **BackLinks** stellt sich komplexer dar. Initialisierung und Verteilung von Nachrichten verlaufen prinzipiell wie beim Link – die Zuordnung bzw. Vermittlung der Antwort macht aber zusätzliche Schritte notwendig. Bevor die Nachricht eines auslösenden Events auf einem BackLink publiziert wird, versieht sie der Event Broker mit einer eindeutigen *CallbackID*. Damit der Event Broker ein „normales“ Ereignis von einem Callback-Event unterscheiden kann, muss letzteres die CallbackID der Ausgangsnachricht mitführen. Dadurch kann die Antwort erkannt, und an die entsprechend Callback-Operation vermittelt werden, wie in Abbildung 5.14 veranschaulicht.
Auch permanente Aktualisierungen über den impliziten Rückkanal sind jeweils mit der entsprechenden ID versehen. Da dem Event Broker die Informationen aus dem Kompositionsmodell zur Verfügung stehen, obliegt ihm die Überprüfung, ob kontinuierliche Updates erlaubt sind (Modellattribut `syncable`) und ob sie die spezifizierten Zeitintervalle nicht unterschreiten (Modellattribut `syncThreshold`).

Die Synchronisation von Eigenschaften über **PropertyLinks** wird auf die oben beschriebenen Link-Mechanismen des Event Brokers abgebildet. Dazu werden die impliziten Change-Events und Setter-Methoden der betroffenen Eigenschaften durch einen Link verknüpft. Der dadurch modellierte Datenfluss bringt jedoch drei Probleme mit sich:

(1) Da die Synchronisation zwischen allen am PropertyLink angeschlossenen Komponenten erfolgt, muss zunächst verhindert werden, dass die auslösende Komponente über ihre eigene Zustandsänderung informiert wird. Dazu wird sie vom Event Broker bei der Verteilung herausgefiltert.

(2) Durch das Setzen neuer Property-Werte in allen angeschlossenen Komponenten schicken diese wiederum Change-Events. Deren Verteilung muss freilich unterdrückt werden, um Endlosschleifen bei der Synchronisation zu vermeiden. Die Lösung hierfür besteht in der Wertprüfung durch den Event Broker. Der zuletzt verteilte Wert wird gespeichert und dient vor der Publikation neuer Change-Events als Referenz. Sind die publizierten Werte gleich, wird die Verteilung abgebrochen.

(3) Die gleichzeitige Publikation bzw. Überschneidung verschiedener Change-Events kann u. U. zum Datenverlust bei der Synchronisation führen. Deshalb wird nach dem FIFO-Prinzip eine Warteschlange aufgebaut: Falls während der Synchronisation eines Property-Wertes von einer der beteiligten Komponenten ein neuer Wert publiziert wird, wird dieser gespeichert. Die Verteilung des neuen Wertes wird allerdings bis zum Abschluss der aktuellen Synchronisation ausgesetzt, und erst danach durchgeführt.

Beispiel: *Die Eigenschaften P_A, P_B, P_C und P_D von vier Komponenten seien über einen PropertyLink verbunden. Ein Change-Event für P_A zeigt dessen neuen Wert „X“ an und wird vom Event Broker zunächst an P_B weitergeleitet. Inzwischen ändert sich P_C auf den Wert „Z“. Das entsprechende Change-Event wird vom Event Broker registriert und der neue Wert gesichert. Dann wird P_D zunächst auf „X“ gesetzt, um die erste Synchronisation abzuschließen (Das Setzen von P_C entfällt aufgrund der zwischenzeitlichen Aktualisierung). Erst dann werden P_A, P_B und P_D über den neuen Wert „Z“ informiert.*

Ein orthogonales Problem stellt die Abbildung plattform- bzw. endgerätespezifischer Interaktionstechniken auf das ereignisorientierte Kommunikationsmodell dar. Wie in Abschnitt 5.2.2.3 bereits angesprochen, nimmt die Interaktion über **Drag-and-Drop** eine Sonderrolle ein, da sie die Abstimmung zwischen Komponenten und Laufzeitumgebung nötig machen. Die Funktionalität zum „Ziehen“ wird i. d. R. durch Komponentenentwickler in der Technologie ihrer Wahl umgesetzt, sodass die komponentenübergreifende Funktionalität bei Komponenten verschiedener Herkunft nicht gegeben ist. Die MRE muss deshalb über den Start der Interaktion informiert werden, um beim „Ablegen“ die entsprechenden Daten an die passenden Operationen vermitteln zu können.

Ausgangspunkt der Interaktion bilden *Data Sources* (Quellen), die aus der Komponentenbeschreibung ersichtlich und auch im Kompositionsmodell repräsentiert sind (vgl. Abschnitt 5.2.2.3). Sie verdeutlichen, welche Daten Komponenten per *Drag* verfügbar machen. Die Ablage dieser Daten wird prinzipiell nur in Komponenten erlaubt, die eine Operation mit der gleichen Signatur wie die Quelle besitzen.

Der Datenaustausch folgt den in Abbildung 6.7 dargestellten Schritten:

① Eine Komponente registriert lokal das „Ziehen“ von Daten.

② Sie informiert die MRE durch ein *Drag*-Event mit Referenz auf die entsprechende Data Source darüber, dass und welche Art von Daten gezogen werden.

③ Die MRE legt unsichtbare Ebenen – *Dropzones* – über alle anderen sichtbaren Komponenten. Je nachdem, ob diese kompatible Operationen enthalten oder nicht, wird die Dropzone aktiviert oder gesperrt.

④ Werden Daten auf einer aktiven Dropzone „fallen gelassen“, wird dies von der MRE registriert. Falls die darunter liegende Komponente mehrere kompatible Operationen anbietet, muss die Auswahl durch den Nutzer erfolgen. Daraufhin werden alle Dropzones entfernt.

⑤ Die MRE ruft von der ausgehenden Komponente die gewählten Daten ab. Deren Abfrage, Verarbeitung und etwaige Mediation finden somit nur im Bedarfsfall statt.

⑥ Die bereitgestellten Daten werden an die Operation der Zielkomponente gesendet.

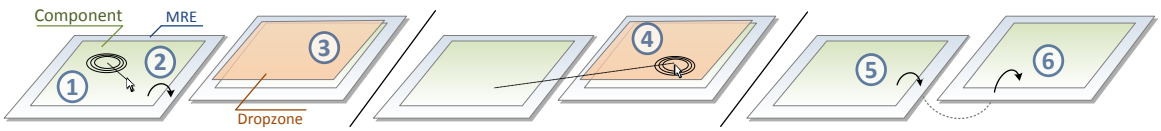

Abb. 6.7: Ablauf der Drag-and-Drop-Interaktion

Grundlegend werden also native Ereignisse zwischen Komponenten und der Laufzeitumgebung vermittelt. Komponenten geben über interne Interaktionen durch Events Auskunft, und die MRE ruft bei externen Interaktionen – in diesem Fall der *Drop* – vordefinierte Methoden der Komponenten auf. Durch diese Vermittlung bleibt die SoC gewahrt, und Komponenten unterschiedlicher Herkunft und Technologie können durch vordefinierte Schnittstellen (Interaktionsereignisse und Operationen zum Datenzugriff) über plattformspezifische Interaktionstechniken miteinander gekoppelt werden. Für die Zielkomponenten bleibt die Drag-and-Drop-Interaktion völlig transparent, da sie in einem konventionellen Operationsaufruf mündet. Das gleiche Konzept kann genutzt werden, um beispielsweise Interaktionen mit Sprachsteuerung oder über *Tangibles* [Shaer und Hornecker, 2010] auf das Event-Modell abzubilden.

Wie bereits ausführlich beschrieben, ist durch das Kompositionsmodell und die Discovery-Algorithmen die semantische Kompatibilität der ausgetauschten Daten sichergestellt. Syntaktische Differenzen auf Signatur- und Datenebene müssen jedoch ausgeglichen werden – egal, um welche Kommunikationsform es sich handelt. Der nächste Abschnitt stellt die entsprechenden Konzepte der MRE vor.

6.2.4.2 Mediation auf Signatur- und Datenebene

Zu Beginn des Kapitels wurde gezeigt, wie Komponenten auf Basis ihrer funktionalen, nicht-funktionalen und Datensemantik gesucht und zur Laufzeit zu einer Mashup-Anwendung komponiert werden können. Da dieser Prozess bewusst auf semantischer

statt syntaktischer Kompatibilität basiert, müssen Mediationsmechanismen bereitgestellt werden, die syntaktische Schnittstellenabweichungen zwischen Komponenten zur Laufzeit überbrücken.
Grundlage für die Mediation stellen die folgenden, bereits vorgestellten Konzepte dar: 1) die semantische Typisierung der ausgetauschten Daten und Kommunikationskanäle, 2) die Transformationsvorschriften der SMCDL und 3) das *MatchingResult*. Sobald eine Komponente auf Grundlage einer Vorlage integriert (oder ausgetauscht) wird, müssen eine oder mehrere der folgenden Aktionen durchgeführt werden:

1. Abbildung bzw. Zuordnung von Properties, Operations, Events und Parametern.
2. Umordnung und/oder Filterung von Parametern der ein/ausgehenden Events.
3. Ausführung der Transformationen aus dem *Binding* der Komponentenbeschreibung (vgl. Abschnitt 5.1.3.2), um Kompatibilität der Daten mit den Standard-*Groundings* zu erreichen – dies gilt für alle Komponenten (unabhängig von der Nutzung von Vorlagen).
4. Umwandlung einer Instanz eines semantischen Konzeptes (in Form eines Event-Parameters) zu einer Instanz einer entsprechenden Superklasse des Konzeptes (*Up-Cast*) im Fall eines *Plugin*-Matches des Parameters (vgl. Abschnitt 6.1.2).

Wie in Abbildung 6.8 zu sehen ist, sind an der Mediation im Wesentlichen zwei Module beteiligt: Auf syntaktischer Ebene sichern *Wrapper* die Schnittstellenkompatibilität von Komponenten. In Anlehnung an das Entwurfsmuster *Adapter* [Gamma et al., 1995] umschließen sie Komponenteninstanzen und bilden ihre Schnittstellen auf die Spezifikation gemäß dem Kompositionsmodell ab. Die Mediation von Daten auf der semantischen Ebene übernimmt der *Mediator*. Er dient der Auflösung der in Abschnitt 6.1.2 beschriebenen, nicht-exakten Übereinstimmungen zwischen verwendeten Datentypen. Beide Module werden im Folgenden vorgestellt.

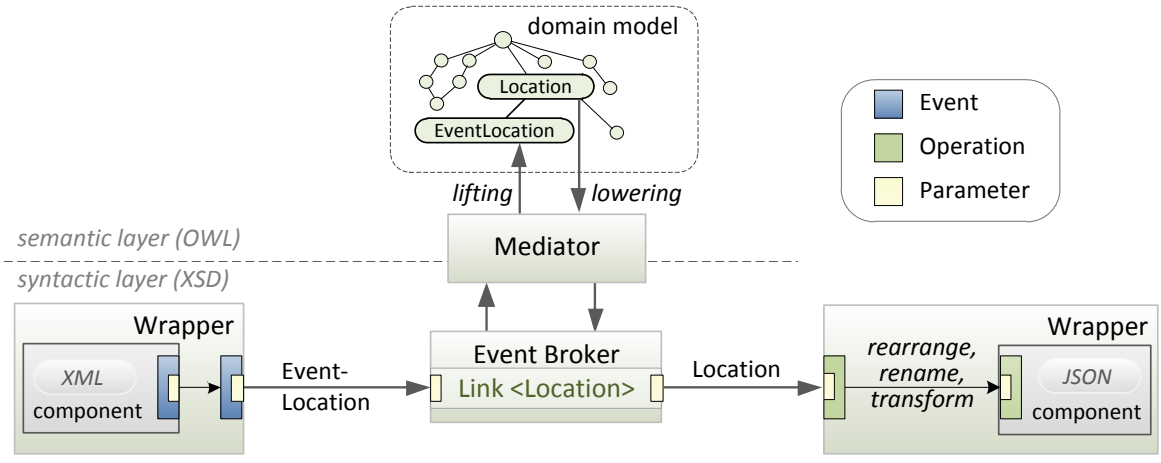

Abb. 6.8: Mediation am Beispiel eines Ein-Parameter-Events

Wrapper bilden das Bindeglied zwischen Komponenten und Event Broker. Aus Sicht der Komponenten agieren sie als *Proxy* [Gamma et al., 1995] und nehmen die Rolle des Brokers ein. Somit können sie gesendete Nachrichten anpassen und erweitern, z. B. um die Komponenten-ID. Für den Event Broker wiederum repräsentieren Wrapper je eine Komponente, d. h. sie entsprechen der Modellvorlage und fangen jegliche

eingehenden Ereignisse ab. Somit können sie zur Isolation von Komponenten, z. B. bei deren Austausch (vgl. Abschnitt 6.3.3), genutzt werden.
Durch ihre Position können Wrapper als Adapter eingesetzt werden, wenn integrierte Komponenten syntaktische Unterschiede zur Vorlage aus dem Kompositionsmodell aufweisen, d. h. sie decken die ersten drei der o. g. Aufgaben ab. Das *MatchingResult*, welches aus der Discovery-Phase hervorgeht, dient dem Wrapper als Anleitung zur Umbenennung, Umsortierung und Filterung von Operationen, Ereignissen und ihren Parametern. Die Schnittstelle einer integrierten Komponente ist nach diesen Anpassungen kompatibel zu der zugrunde liegenden Vorlage. Durch die Anwendung der Transformationen aus dem *Binding* wird zudem sichergestellt, dass alle Daten ausgehender Events den vordefinierten *Grounding*-Schemata entsprechen, während eingehende Parameter auf komponenteninterne Datenmodelle abgebildet werden.
Im Beispiel aus Abbildung 6.8 sendet die linke Komponente ein Ereignis mit einem Parameter vom Typ *EventLocation*. Dieser liegt bereits in XML-Form entsprechend dem Grounding vor, sodass die Nachricht vom Wrapper zum Event Broker weitergeleitet wird. Der Link zur Publikation besitzt als Signatur einen Parameter vom Typ *Location*. Zur Weiterleitung der Nachricht an die angeschlossenen Operationen muss das Datum somit von einer *EventLocation* in eine *Location* umgewandelt werden. Dazu greift der Broker auf die Funktionalitäten des Mediators zurück, der den Up-Cast wie unten beschrieben durchführt. Das resultierende Datum wird nun über den Link weiter verteilt. Der Aufruf der Zieloperation, z. B. showLocation erfolgt über den Wrapper der zweiten Komponente. Letztere unterstützt intern jedoch nur die Methode setMarker und erwartet den Parameter als JSON-Objekt. Die Abbildung erfolgt durch den Wrapper, wobei die Zuordnung zum anderen Operationsnamen aus dem *MatchingResult* ersichtlich ist, und die Datentransformation auf Basis der Informationen aus dem *Binding* erfolgt.

Die semantische Mediation über *Up-Casts* erfolgt durch den **Mediator**, ein Modul der MRE. Wie in Abbildung 6.8 erkennbar ist, erfolgen die Entscheidung und Ausführung der semantischen Mediation auf den Kommunikationskanälen, initiiert durch den Event Broker. Die Gründe hierfür liegen – im Gegensatz zur Mediation durch die Wrapper – vor allem in der gesteigerten Effizienz und Performanz. Falls mehrere Empfänger auf einem Kanal den gleichen Up-Cast benötigen, muss dieser nur einmal pro Kanal vorgenommen werden. Außerdem ist so die Vereinigung mehrstufiger Up-Casts (auf Sender- und Empfängerseite) möglich.
Vor der semantischen Mediation werden die XSLT-Skripte für Lifting- und Lowering vom TyRe abgerufen (Abschnitt 6.2.1). Danach erfolgt die Umwandlung in drei Schritten, die den Prinzipien von SAWSDL (Abschnitt 3.1.2) folgen. Zunächst werden die Instanzdaten einer Ausgangsklasse Q vom Grounding – der standardisierten XML-Serialisierung – in einen RDF-Graphen überführt (*Lifting*). Sie entsprechen dann einem Individuum der Klasse Q des Domänenmodells. Dieses wird durch einen Up-Cast in ein Individuum der Ziel-Klasse Z umgewandelt, indem Q aus Sicht von Z über eine SPARQL-Anfrage angefordert wird. Das Ergebnis im SPARQL Query Results XML Format kann schließlich durch das Lowering wieder in die syntaktische Form transformiert werden. Es entspricht dann dem Grounding des Zieltyps Z. Vor der Weiterleitung an die Zielkomponente bzw. dessen Wrapper werden durch den Broker etwaige Abbildungsregeln zwischen Teilnehmern eines Kommunikationskanals (*ParameterMappings*, vgl. Abschnitt 5.2.2.2) umgesetzt.

Durch die Nutzung von SPARQL beim Up-Cast wird die Unabhängigkeit von den verschiedenen Variationen von RDF/XML gewahrt, wie im SAWSDL Usage Guide empfohlen [@Akkiraju und Sapkota, 2007]. Im Gegensatz zum SAWSDL werden im hier vorgestellten Ansatz jedoch Liftings und Lowerings nur dann eingesetzt, wenn Subklassenbeziehungen aufgelöst werden müssen. Anderenfalls werden Daten auf syntaktischer Ebene ausgetauscht. In diesem Fall nimmt der Event Broker eine Typprüfung vor, um die Validität der ausgetauschten Daten sicherzustellen.
Abbildung 6.8 zeigt einen typischen Anwendungsfall der Mediation, bei dem die spezifische Repräsentation eines Ortes vom Typ EventLocation in eine generische Instanz vom Typ Location umgewandelt werden muss, um von der Empfängerkomponente verarbeitet werden zu können. Das Domänenmodell dient dabei als Bindeglied, um die Instanzdaten entlang des Vererbungsgraphen per „Cast“ nach oben zu reichen. Der Mediator führt dazu ein Lifting des Ortes in RDF/XML, den Cast auf den Typ Location sowie das Lowering in die Grounding-Repräsentation durch.

Alles in allem ermöglichen die vorgestellten Mediationskonzepte die dynamische Integration semantisch kompatibler Komponenten, solange deren genutzte Datentypen auf semantischer Ebene mediierbar sind. Grundlage hierfür bieten gemeinsame Domänenmodelle, über die syntaktische Unterschiede durch Auflösung von Subsumptionsbeziehungen überbrückt werden. Dies impliziert die Nutzung einheitlicher, vordefinierter Ontologien – das Konzept ist jedoch nicht hierauf beschränkt, sondern kann durch *Ontology-Mapping*-Verfahren erweitert werden, um die Mediation beliebiger Daten zu unterstützen. Einen guten Überblick dazu bieten Euzanat und Shvaiko (2007).
Damit ist die Vorstellung der wichtigsten Konzepte zur dynamischen Integration und Ausführung universeller Kompositionen abgeschlossen. Das folgende Unterkapitel widmet sich nun ihrer kontextabhängigen Anpassungen zur Laufzeit.

6.3 Unterstützung von Adaption zur Laufzeit

In den letzten Abschnitten wurden Konzepte zur kontextsensitiven Integration von Mashup-Komponenten und deren koordinierten Ausführung vorgestellt. Im Weiteren liegt der Schwerpunkt auf der dynamischen Adaption einer solchen kompositen Anwendung entsprechend dem in Abschnitt 5.3.2 vorgestellten *Adaptivity Model*. Zur Unterstützung der beschriebenen Adaptionstechniken wurde eine Adaptionsinfrastruktur entwickelt, welche mit bestehenden Kompositionssystemen gekoppelt werden kann und die aspektorientierte Adaption auf Komponenten- und Kompositionsebene ermöglicht. Wie Abbildung 6.9 zeigt, ist das Adaptionssystem in zwei Bereiche für die Adaptions- und Kontextverwaltung geteilt.
Die **Adaptationsverwaltung** (links) ist mit der MRE gekoppelt und realisiert die Auswertung und Umsetzung der formulierten Adaptionsaspekte, die als generische ECA-Regeln interpretiert werden. Letztere verweisen auf spezifische Teile einer Komposition (Komponenteneigenschaften und -instanzen, Layouts, usw.), die in Folge von Kontextveränderungen (*Events*) und in Abhängigkeit von optionalen Bedingungen (*Conditions*) durch Adaptionsaktionen (*Actions*) anzupassen sind. Die dynamische Auswertung der Regeln erfolgt durch die *Rule Engine*, welche bei Bedarf den Adaptation Manager zur Ausführung der Aktionen veranlasst. Letztere

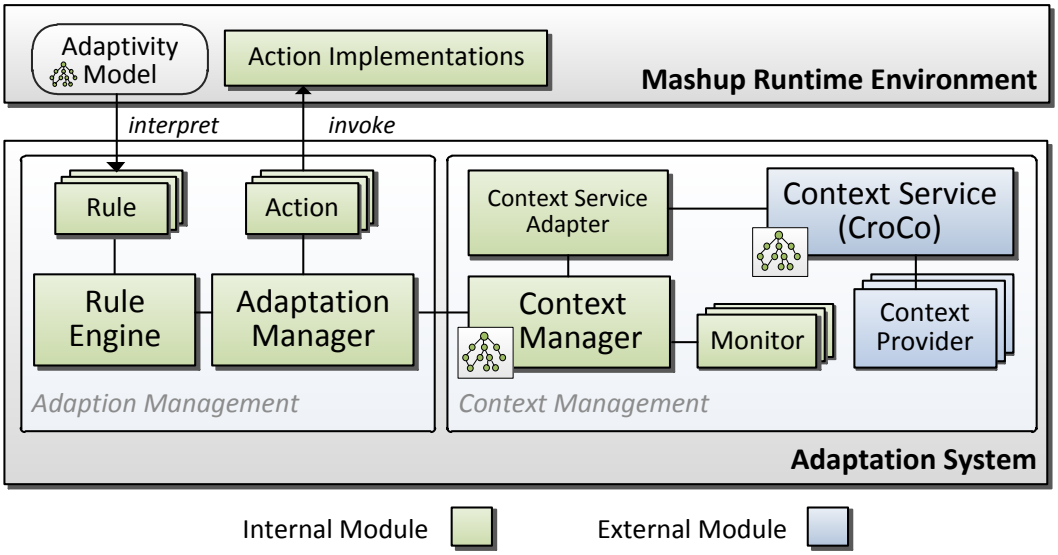

Abb. 6.9: System zur dynamischen Adaption kompositer Mashup-Anwendungen

entsprechen den in Abschnitt 5.3.1 vorgestellten Techniken, für welche je nach Kompositionsplattform verschiedene Implementierungen existieren können.
Der Bereich der **Kontextverwaltung** (rechts) beinhaltet die Module zur Kontextüberwachung und -modellierung. Die Überwachung wird von *Monitoren* oder unabhängigen *Context Providern* übernommen. Die Kontextmodellierung und -verarbeitung (Schlussfolgerung, Konsistenzsicherung, etc.) erfolgt durch ein dediziertes System, da die hohe Komplexität dieser Aufgaben die Auslagerung in einen Dienst (*Context Service*) nahelegt. Zur Verbesserung von Performanz und Netzwerklast verwaltet der *Context Manager* eine lokale Teilkopie des Kontextmodells, welches permanent mit dem entfernten Kontextdienst synchronisiert wird. Durch die Nutzung verschiedener *Context Service Adapter (CSA)* ist die Unabhängigkeit vom Kontextdienst gegeben.
Die folgenden Abschnitte beleuchten die Bestandteile der Adaptionsinfrastruktur im Hinblick auf die Überwachung und Modellierung von Kontextinformationen sowie die Umsetzung der durch das Modell definierten Adaptionstechniken näher.

6.3.1 Kontexterfassung, -modellierung und -verwaltung

In dieser Arbeit steht die Kontextnutzung bei der Adaption kompositer Anwendungen zur Laufzeit im Vordergrund. Auch wenn die Überwachung und Modellierung der dafür nötigen Kontextdaten nicht den Kern der Konzeption bilden, so muss doch betrachtet werden, in welcher Art und Weise Kontextwissen in die Kompositionsinfrastruktur und den Adaptionsprozess eingebunden wird.
Bei der Vorstellung der Grundlagen wurde hervorgehoben, dass die Kontextverwaltung aufgrund ihrer hohen Komplexität zunehmend von kontextsensitiven Systemen entkoppelt wird. Dies bietet sich auch für die vorgestellte Kompositionsinfrastruktur an. Hier kann der Belang der Kontextverwaltung als Dienstleistung gekapselt werden, die verschiedenen Plattformen und Anwendungen zur Verfügung steht. Sie kann somit unabhängig skaliert, gewartet und erweitert werden, und es ist von einer höheren Qualität und Aktualität der Kontextdaten auszugehen, da mehr Clients an ihrer Überwachung beteiligt sind.

Aus der Sicht der Adaptionsplattform bietet die Unabhängigkeit vom Kontextdienst den Vorteil, dass letzterer jederzeit gegen Alternativen ausgetauscht werden kann. Neben der Bindung alternativer Dienste im Fehlerfall ist ebenso die Nutzung einer eigenen bzw. plattformspezifischen Lösung möglich, ohne dass das Adaptionssystem angepasst werden muss. Die vollständige Entkopplung wird allein durch einen entsprechenden CSA gewährleistet.
Die beiden folgenden Abschnitte behandeln die Konzepte zur Kontextverwaltung im Hinblick auf den unabhängigen Kontextdienst (Abschnitt 6.3.1.1) und das Adaptionssystem (Abschnitt 6.3.1.2).

6.3.1.1 Der ontologiebasierte Kontextdienst CroCo

Für die Belange der Kontextverwaltung wurde der ontologiebasierte Kontextdienst Cross-Application Context Management (CroCo) konzipiert. Er bietet *Context Providern* eine Schnittstelle zum Registrieren und Aktualisieren von Kontextdaten und erlaubt *Context Consumern* den Abruf ebensolcher.
Ein Vorteil dieser Entkopplung besteht in der anwendungs- und plattformübergreifenden Nutzung: Software- und Hardwaresensoren verschiedenster Systeme können gemeinsam zum Aufbau des Kontextmodells beitragen, und gleichermaßen kann CroCo als Kontextlieferant für beliebige kontextadaptive Anwendungen und Plattformen dienen. Die Vielzahl möglicherweise redundanter Kontextmonitore fördert die Aktualität und so implizit die Qualität der bereitgestellten Kontextdaten.

Kontextmodell

Innerhalb von CroCo werden Kontextdaten durch Instanzen eines generischen Kontextmodells – der CroCo Ontology (CroCoOn) – repräsentiert. Es besteht aus diversen Teilmodellen, die generische Aspekte wie Zeit, Ort, Nutzer und Endgeräte auf einer abstrakten, anwendungsunabhängigen Ebene beschreiben, wobei auf Konzepte aus etablierten Ontologien, wie SOUPA [Chen et al., 2004] und PROTON [@SEKT, 2005] zurückgegriffen wird.
Diese generischen Konzepte können spezialisiert werden, um domänenspezifisches Wissen zu modellieren. Abbildung 6.10 veranschaulicht die Erweiterung anhand einiger exemplarischer Modellentitäten aus dem Umfeld der Anwendungsszenarien aus Abschnitt 2.2. Es wird eine neues Konzept UserProfile definiert, welches im Bezug zu einer Person steht und dieser Vorlieben (*UserPreferences*) sowie Hard- und Software-Eigenschaften zuweist. Das Konzept *WebBrowserConfig* kann später u. a. dafür genutzt werden, Informationen über installierte Plug-ins zu erhalten. Die Eigenschaft *currentLocation* eines Nutzers gibt Auskunft über dessen Aufenthaltsort, welcher eine Spezialisierung des generischen Konzeptes „Stadt“ (City) ist.
Die derartige Erweiterung der Kontextbasis erfolgt in CroCo über *Profile*, die neben den domänenspezifischen Modellen auch Regeln bzw. Fakten und Sicherheitsrichtlinien enthalten können, wie in Abbildung 6.11 links zu sehen ist.
Die derartige ontologiebasierte Modellierung von Kontextdaten kommt inzwischen in der Mehrzahl kontextsensitiver Systeme zum Einsatz [Baldauf et al., 2007]. Sie ermöglicht u. a. die Nutzung etablierter Domänenmodelle, die automatisierte Konsistenzsicherung und Verarbeitung sowie die Ableitung neuen Kontextwissens durch semantische Inferenzmechanismen.

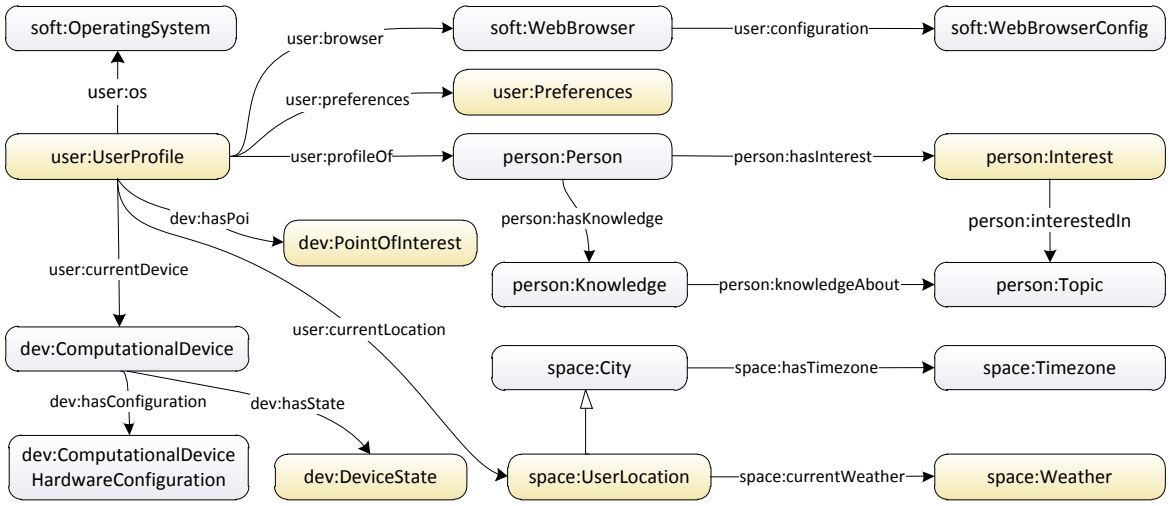

Abb. 6.10: Erweiterung der generischen Kontext-Ontologie am Beispiel

Ein typisches Beispiel für die Notwendigkeit solch impliziten Wissens stellt die Konfiguration von Komponenten bzw. Inhalten in Abhängigkeit von der *Volljährigkeit* von Nutzern dar, die sich je nach Alter und Staatsangehörigkeit deutlich unterscheiden. Auf Basis von Fakten eines semantischen Modells kann das Konzept jedoch automatisch geschlussfolgert werden. Anwendungen bleiben indes unabhängig von derartigen Berechnungen, d. h. Konfigurationsparameter und Adaptationsaspekte können direkt auf solch implizite Kontextdaten verweisen, deren Ermittlung in der Verantwortung des von CroCo liegt.

Die Referenzierung von Kontextdaten erfolgt somit im vorliegenden Konzept, d. h. im Kompositionsmodell (Abschnitt 5.3) und in den Sortierungsregeln (Abschnitt 6.1.3.1), durch Verweise auf semantische Konzepte. Eine Abbildung auf andere Kontextrepräsentationen bzw. -dienste ist bei der Auswertung der Adaptionsaspekte durch die CSA ebenfalls möglich.

Architektonischer Überblick

Abbildung 6.11 zeigt den Aufbau von CroCo schematisch. Er folgt dem *Blackboard*-Modell, einer datenzentrierten Sicht, bei der alle beteiligten Parteien Kontextinformationen auf das *Blackboard* schreiben, davon lesen, oder sich für die Benachrichtigung bei Änderungen registrieren. Die Kontextverwaltung bleibt somit unabhängig von der Anzahl und Art der angeschlossenen Bereitsteller und Konsumenten von Kontextdaten. In der Abbildung lassen sich anhand der Schichtung die drei wesentlichen Aufgaben von CroCo unterscheiden:

Im Zentrum steht die **Verwaltung von Kontextdaten** (*Context Data Management*) durch den *Context Store*. Dieser arbeitet auf Instanzen der generischen und domänenspezifischen Konzepte aus *CroCoOn* und den Domänenprofilen, die durch den *Profile Store* bereitgestellt werden. Bei der Verwaltung wird zwischen *Consistent Context* und *Inferred Knowledge* unterschieden. Ersterer repräsentiert das aktuell valide, semantische Kontextmodell, welches sich aus Aktualisierungen der Context Provider ergibt. Letzterer enthält impliziten Kontext, der auf Basis der verfügbaren Fakten und Regeln geschlossen werden kann. Für Context Consumers ist diese Trennung unsichtbar – sie greifen immer auf den Context Store zu. Nur die *Context*

History bleibt für sie verborgen. Sie enthält alle eingehenden Änderungen am Kontextmodell und wird u. a. für statistische Analysen, z. B. zur Veränderlichkeit von Kontextdaten und Verlässlichkeit von Providern, und für das Reasoning benötigt.

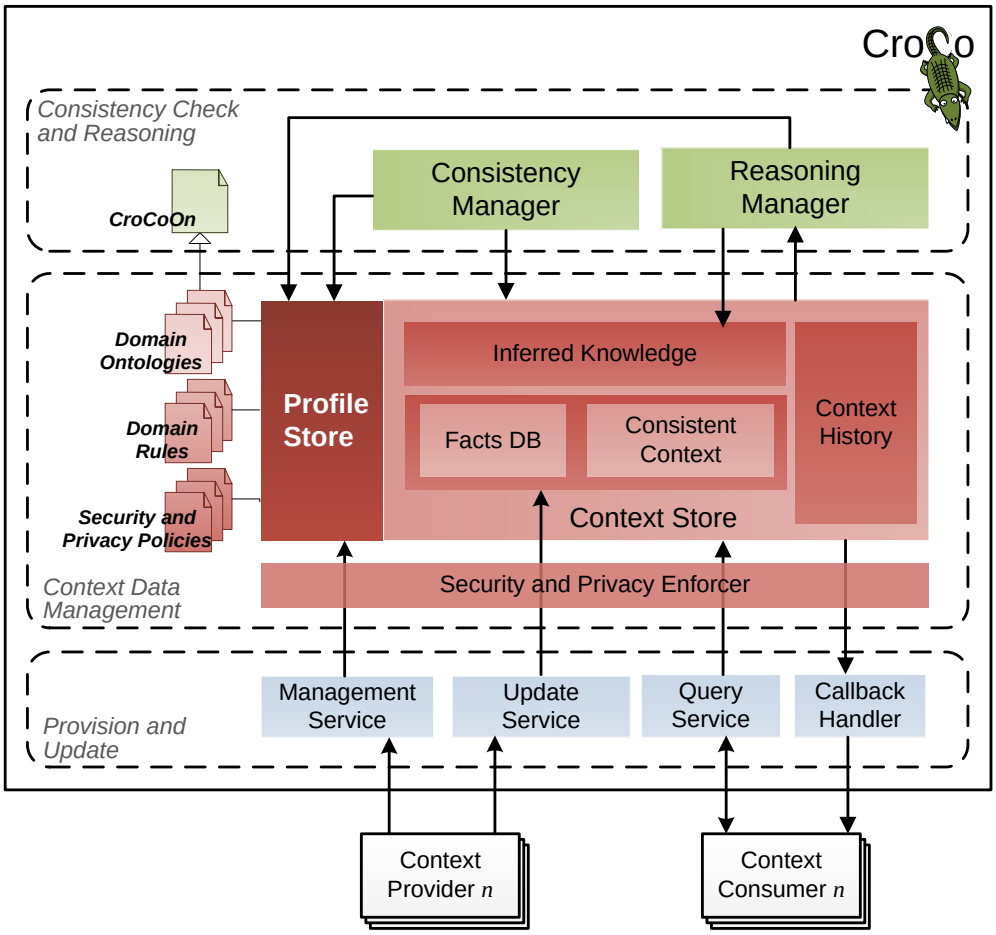

Abb. 6.11: Aufbau des Kontextverwaltungsdienstes CroCo

Die **Konsistenzprüfung** des Kontextmodells erfolgt durch den *Consistency Manager* auf Basis der verfügbaren Regeln und Fakten, sobald neue Kontextdaten eingehen. Dazu sind bei ihm eine Reihe von Modulen registriert, die jeweils einen bestimmten Aspekt, z. B. Datentypen oder Kardinalitäten von Modellinstanzen, prüfen.

Die **Schlussfolgerung von implizitem Wissen** – dem *Inferred Knowledge* – übernimmt der *Reasoning Manager*, welcher bei ihm registrierte Reasoner koordiniert. Jede Änderung am konsistenten Basismodell führt zum Aufruf der Reasoner, die ihrerseits auf die Fakten der *Facts DB* zugreifen. Letztere zerfallen in verschiedene Korpora, z. B. in grundlegende und spezifische Fakten der jeweiligen Domänenprofile. Bevor das gefolgerte Wissen schließlich in den Context Store eingeht, wird es einer erneuten Konsistenzprüfung unterzogen.

Die **Aktualisierung und Bereitstellung** von Kontextdaten wird durch zwei Dienste sichergestellt. Der *Update Service* erlaubt die Aktualisierung von Kontextdaten, und der *Query Service* deren synchrone Abfrage. Über einen *Callback Handler* ist auch die Registrierung für Kontextdaten möglich, deren Änderung im Modell dann zur Benachrichtigung aller registrierten Context Consumer führt.

Der Zugriff auf Daten von außen kann durch eine Sicherheitsschicht (*Security/Privacy Enforcer*) beschränkt werden. Dazu kommen Policies zum Einsatz, die z. B. über Domänenprofile in CroCo eingebracht werden können. Zur Registrierung und Aktualisierung dieser Profile wird schließlich ein *Management Service* angeboten.
Mehr Details zur Funktionsweise und dem anwendungsübergreifenden Einsatz, z. B. in den Projekten K-IMM [Mitschick, 2010] und VCS [Niederhausen et al., 2010] werden durch Mitschick et al. (2010) erläutert.

Für die im letzten Teilkapitel vorgestellte Kompositionsinfrastruktur bedeutet der Einsatz von CroCo eine Erleichterung von den aufwändigen Belangen der Kontextverwaltung. Die Verantwortung der Validierung, Konflikterkennung und die Ableitung zusätzlicher Kontextinformationen kann von der Kompositionsumgebung entkoppelt werden, wie in Abschnitt 2.3 gefordert. So ist sichergestellt, dass sich sowohl der kontextsensitive Integrationsprozess als auch das Adaptionssystem auf ein jederzeit valides und konsistentes Kontextmodell stützen können.
Die Nutzung von CroCo bleibt freilich konzeptionell freigestellt. Als Teil der Referenzarchitektur und der prototypischen Umsetzung wird er in den folgenden Abschnitten jedoch synonym für den verwendeten Kontextdienst behandelt.

6.3.1.2 Kontexterfassung und -verwaltung als Teil der Laufzeitumgebung

Aus Sicht von CroCo stellt die MRE einen *Context Consumer* dar, der die Kontextmodellinstanzen zur Anpassung von kompositen Anwendungen nutzt. Gleichzeitig übernimmt es die Rolle eines *Context Providers*, der u. a. Auskunft über Browsereigenschaften und Nutzerinteraktionen liefert.
Eine lokale Kontextverwaltung durch die Laufzeitumgebung ist nötig, um vom genutzten Kontextdienst – in diesem Fall CroCo – unabhängig zu bleiben und eine stabile und hohe Performanz zu gewährleisten. Anwendungsspezifische Teile des entfernten Kontextmodells werden deshalb lokal gehalten und fortwährend mit dem Dienst synchronisiert. Nicht jeder Aufruf eines Kontextparameters muss somit weitergeleitet werden – er kann bereits lokal verarbeitet und ausgewertet werden. Die folgenden Abschnitte stellen die Bestandteile der lokalen Kontextinfrastruktur der Laufzeitumgebung dar (vgl. Abbildung 6.9).
Kontext-Monitore sind Software-Komponenten des Adaptionssystems, welche Kontextänderungen erfassen und an die Umgebung weiterleiten. Intern bilden sie rohe Sensordaten auf Entitäten des genutzten Kontextmodells ab. Für das vorliegende Konzept und die passende Implementierung entspricht dies der Erstellung von Instanzen der semantischen Modellkonzepte. Zusätzlich können sie sog. *Confidence*-Werte mitliefern (vgl. [Pietschmann et al., 2008]), die die Aussagewahrscheinlichkeit der Messung darstellen. Die Konfiguration der Monitore kann anwendungsspezifisch erfolgen und sich beispielsweise auf Schwellwerten oder zeitliche Beschränkungen der Aktualisierungen beziehen. In ihrer Gesamtheit müssen die Kontext-Monitore nicht alle von Anwendungen benötigten Kontextparameter abdecken – aus dieser Motivation heraus erfolgte die Auslagerung der Kontextverwaltung. Es muss davon ausgegangen werden, dass das Kontextmodell durch andere *Provider* und durch die beschriebenen Reasoning-Mechanismen erweitert wird.
In Anlehnung an das Kontext-Framework von Hinz et al. (2007) wurde die Möglichkeit geschaffen, der Plattform neue Monitore durch Implementierung einer dedizierten

Schnittstelle hinzuzufügen. Gleichzeitig können Komponenten der Komposition diese Aufgabe übernehmen, indem Sie auf die öffentliche Schnittstelle des Context Managers zugreifen. Ist der Bedarf an bestimmten Kontextdaten zur Entwicklungszeit bekannt, so kann die Verfügbarkeit der Kontextinformationen durch das Hinzufügen einer entsprechenden Komponente sichergestellt werden.
Der **Context Manager** bietet den Modulen der MRE und den Mashup-Komponenten eine öffentliche Schnittstelle – unabhängig vom genutzten Kontextverwaltungsdienst – für Ablage und Abruf von Kontextinformationen. In seiner Verantwortung liegt die lokale Verwaltung der Kontext-Wissensbasis, um wiederholte, redundante Anfragen an den Kontextdienst vermeiden zu können. Dazu erfolgt beim Anwendungsstart ein Abgleich mit dem entfernten Kontextmodell, bei dem die aus dem Kompositionsmodell referenzierten Parameter (zur kontextabhängigen Konfiguration und zur Auswertung von Adaptionsaspekten) abgerufen werden. Parallel erfolgt die Registrierung für asynchrone Änderungsbenachrichtigungen über den o. g. Callback Handler. Bei jeder Änderung wird der Context Manager informiert, was zur Erzeugung entsprechender Kontext-Events und somit ggf. zur Auslösung zugehöriger Adaptionsaspekte führt. Auch die Verwaltung der lokalen Kontext-Monitore übernimmt der Manager. Er leitet ihre Daten zur Validierung und späteren Persistierung an den Kontextdienst weiter. Da, wie bereits angesprochen, das Adaptionssystem unabhängig vom genutzten Kontextdienst bleiben soll, erfolgt dessen Anbindung über Adapter.
Die **Context Service Adapter (CSA)** verbergen Spezifika der Implementierung und API der genutzten Kontextdienste hinter einer einheitlichen Schnittstelle. Dadurch wird es möglich, die Kontextverwaltungsinstanz durch eine beliebige entfernte oder auch eigene Lösung zu ersetzen, sofern ein passender Adapter existiert. Für die SoC sprechen verschiedene Gründe: Im Vordergrund steht die getrennte Entwicklung beider Systeme, aber auch Sicherheitsaspekte, z. B. kann der Einsatz im geschäftlichen Umfeld für eine firmeneigene Kontextlösung sprechen. Die Aufgaben der Adapter entsprechen den für das gleichnamige Entwurfsmuster (vgl. [Gamma et al., 1995]) üblichen: Sie realisieren die Abbildung der internen Kontextrepräsentation auf die API des Kontextdienstes und nehmen dabei Datentypumwandlungen vor. Beim Einsatz von CroCo entspricht dies der Erstellung von RDF-Tripeln aus Kontext-Events und umgekehrt.

Mit der Kontextverwaltung ist der grundlegende Teil der Kontrollschleife kontextsensitiver Systeme abgedeckt. Der folgende Abschnitt widmet sich der Darstellung der Funktionsweise einer Adaption im Zusammenspiel von Kontext- und Adaptionsverwaltung sowie Laufzeitumgebung.

6.3.2 Ablauf der dynamischen Adaption

Im Folgenden werden die Schritte zur Adaption einer kompositen Anwendung zur Laufzeit aufgezeigt. Abbildung 6.12 veranschaulicht die daran beteiligten Module und den Ablauf von der Erfassung von Kontextänderungen bis zur Anpassung. Grundsätzlich können zwei Fälle unterschieden werden: Im einfachen Fall werden Kontextaktualisierungen vom Kontextdienst übermittelt, was zur Adaption führt. Im erweiterten Fall gehen diese Aktualisierungen auf Kontextänderungen zurück, die durch lokale Monitore registriert wurden. In Abbildung 6.12 wird der erweiterte Fall dargestellt – die „einfache“ Verarbeitung beginnt mit Schritt sechs.

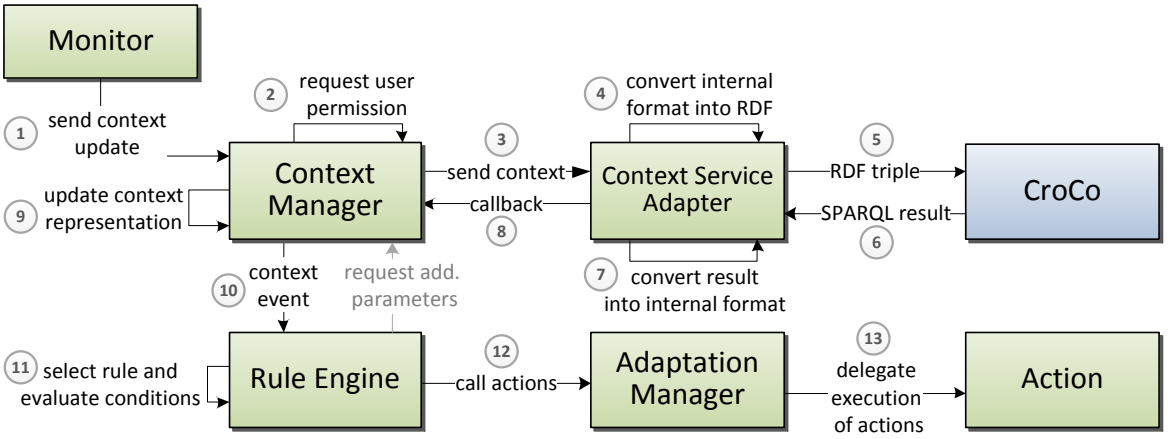

Abb. 6.12: Konzeptioneller Ablauf der dynamischen Anwendungsadaption

1. Ein Kontextmonitor registrierte eine Kontextänderung und sendet diese Information an den Context Manager.
2. /3. Der Context Manager überprüft, ob diese Daten zum externen Kontextdienst publiziert werden dürfen und holt ggf. die Einwilligung des Nutzers ein. Bei Einwilligung oder Unbedenklichkeit der Daten werden diese dem CSA übergeben.
4. /5. Dieser liest die Daten ein, transformiert sie falls nötig in eine für den Kontextdienst kompatible Repräsentation, und sendet sie. Für die prototypische Umsetzung mit CroCo übernimmt der CSA die Umwandlung der Kontextdaten von der lokalen JSON-Repräsentation in RDF/XML.
6. CroCo führt für die eingehenden Daten Konsistenzchecks, Validierung und Reasoning aus. Danach werden alle registrierten Kontextkonsumenten über die Aktualisierung der Daten benachrichtigt. Folglich erhält auch der CSA die Informationen als SPARQL Result, da er initial für alle Parameter registriert wird, die für die Auswertung von Adaptionsregeln nötig sind.
7. /8. Die aktualisierten Kontextdaten werden in das interne Format der Laufzeitumgebung umgewandelt (im Fall einer clientseitigen MRE beispielsweise JSON), und an den Context Manager übergeben.
9. /10. Letzterer aktualisiert seine lokale Repräsentation des Kontextmodells und informiert die Rule Engine über die Änderungen durch *Context Events*.
11. Die Rule Engine analysiert die Events und sucht nach Adaptionsregeln, für die die Kontextänderungen Auslöser darstellen. Für alle entsprechenden Regeln werden die Bedingungen überprüft. Falls dazu weitere Kontextparameter nötig sind, werden sie vom Context Manager bezogen.
12. Als Ergebnis der Überprüfung wird eine Liste mit anzuwendenden Adaptionsaktionen zum Adaptation Manager übergeben.
13. Dieser wählt zu der abstrakten Aktion die plattformspezifische Implementierung aus, startet diese und koordiniert ihre Ausführung.

Obwohl Kontextänderungen der lokalen Monitore direkt in das lokale Modell integriert werden könnten, wurde davon Abstand genommen. Zur Sicherstellung von Qualität und Konsistenz der Kontextdaten werden diese stattdessen zunächst zum

Kontextdienst geleitet. Dieser kann durch seine Mechanismen zur Validierung, zur Konsistenzsicherung und zum Reasoning sicherstellen, ob und wie die Daten in das Kontextmodell zu integrieren sind. Auch im Hinblick auf die Erweiterung in Richtung geräteübergreifende Plattformen bzw. Anwendungen ist dieser Weg vorteilhaft. Zudem wird so dafür gesorgt, dass der Context Manager selbst nur solche Daten verwaltet, die für die Auswertung von Adaptionsregeln benötigt werden.
Die auszuführenden Adaptionsaktionen können in abstrakte Klassen und deren Implementierungen unterteilt werden. Die Entscheidung zur Adaption erfolgt durch die Auswahl einer abstrakten Aktion bei der Regelauswertung. Jeder der in Abschnitt 5.3.1 vorgestellten Adaptionstechniken der Komponenten- und Kompositionsebene ist eine entsprechende Aktion zugeordnet. Für jede muss wiederum eine plattformspezifische Implementierung existieren, die die Anpassung durchführt. Der Komponententausch zur Laufzeit stellt dabei eine besondere Herausforderung dar. Die damit verbundenen Konzepte werden im nächsten Abschnitt vorgestellt.

6.3.3 Dynamischer Austausch von Komponenten

Der Austausch von Komponenten zur Laufzeit stellt sich für interaktive Mashup-Kompositionen deutlich komplexer als für klassische Dienst-Kompositionen dar. Letztere gehen von zustandslosen Bestandteilen aus, die somit einfach blockiert und ausgetauscht werden können. Das in dieser Arbeit vorgestellte Mashup-Komponentenmodell weist jedoch einen Zustand auf, der gesichert und wiederhergestellt werden muss. Während des Austauschs einer Komponente muss zudem dafür gesorgt werden, dass für sie bestimmte Daten nicht verloren gehen. Nur so kann die Konsistenz der Anwendung sichergestellt werden.
Die folgenden Abschnitte stellen die damit verbundenen Konzepte für die Isolation von Komponenten, ihren Austausch und den Zustandstransfer vor.

6.3.3.1 Sicherstellung der Isolation durch Wrapper

Zur Sicherstellung der Isolation beim Komponententausch kommt das Adapter-Konzept zum Einsatz, welches bereits im Zusammenhang der Mediation (Abschnitt 6.2.4) vorgestellt wurde. Es kommt dem Austausch sehr entgegen, da die auszutauschende Instanz durch den Wrapper vollständig verdeckt wird und somit problemlos von der Kommunikation der Anwendung entkoppelt werden kann.
Im Fall eines Austauschs muss der Wrapper blockiert werden können, d. h. ein- und ausgehende Nachrichten werden vom ihm in eine Warteschleife eingereiht. Sie gehen somit nicht verloren, was die Konsistenz und Integrität der Gesamtanwendung sicherstellt, sondern werden in der Reihenfolge ihres Eingangs weitergereicht, sobald der Austausch abgeschlossen ist. Das Verfahren wird gleichsam für die Zeitspanne zwischen Integration und Initialisierung einer Komponente (vgl. Abschnitt 6.2.3.2) angewendet.

6.3.3.2 Ablauf des Komponententauschs

Der Komponententausch beginnt mit dem in Abschnitt 6.1 beschriebenen Discovery-Prozess. Die integrierte Komponente wird dazu als Template aufgefasst – unabhängig davon, ob sie als solche im Kompositionsmodell ausgewiesen ist – und dient als

Ausgangspunkt für die Suche nach Alternativen. Die vorliegende Komponente wird der in Abschnitt 6.1.1 erwähnten Blacklist hinzugefügt, sodass sie von der Suche ausgeschlossen wird. Eine Alternative bestünde in der Filterung durch die MRE selbst, was faktisch aber zur Verringerung der Kandidatenmenge und zu unnötigem Aufwand in der Discovery führen würde. Die Integration des besten Kandidaten selbst läuft ebenfalls ähnlich dem bereits geschilderten Verfahren. Es sind allerdings Erweiterungen nötig, um die Atomarität und Isolation der Aktion sicherzustellen und den Zustandstransfer zwischen alter und neuer Komponente zu ermöglichen.
Das dazu entwickelte Verfahren orientiert sich an den Ansätzen von Irmert et al. (2008) und Mukhija und Glinz (2005) aus dem CBSE-Umfeld, die die Nutzung eines Stellvertreters vorsehen. Dieser dient in der vorgestellten Architektur somit nicht nur der Steigerung der Interoperabilität, wie in Abschnitt 6.2.4 beschrieben, sondern auch der Sicherung des Komponentenzustands zum Beginn und der Isolation während des Austauschprozesses.
Für den Austausch selbst kommt ein erweitertes Phasen-Commit-Protokoll zum Einsatz. Der Austausch einer Komponente K_A gegen K_B, z. B. angestoßen durch einen Adaptionsaspekt oder eine Nutzerinteraktion, durchläuft die folgenden Phasen (an passender Stelle wird jeweils auf den Zustand $S_{A/B}$ der Komponenten gemäß dem in Abschnitt 6.2.3.1 beschriebenen Lebenszyklus verwiesen):

1. Block Vor der Ausführung jeglicher Adaption wird durch den Adaptation Manager eine Synchronisationssperre aktiviert. Dadurch wird die Ausführung weiterer Adaptionen bis zur Fertigstellung des Austauschs ausgesetzt, um die Isolation zu gewährleisten und mögliche Wechselwirkungen auszuschließen.

2. Prepare In dieser Phase wird der Austausch von K_A ($S_A = Interactive$) vorbereitet.

1. Zunächst wird der Wrapper W_A der Komponente K_A blockiert. Damit leitet er keine Nachrichten von außen mehr an K_A weiter.
2. Die Methode prepare von K_A wird durch den Koordinator – im Normalfall die Aktion – aufgerufen. Sie signalisiert der Komponente den Austausch und gibt ihr die Möglichkeit, ausstehende innere Berechnungen oder Transaktionen abzuschließen, um einen sicheren Zustand zu erreichen.
3. K_A signalisiert den Abschluss der Aktionen durch ein entsprechendes *Life-Cycle*-Event (vgl. Abschnitt 6.2.3.1), welches durch den Koordinator interpretiert wird. Es kann ebenfalls zur Anzeige von Fehlern oder zur Zeitüberschreitung kommen.

3. Check Das Ergebnis der Prepare-Phasen bestimmt den weiteren Verlauf:

a Falls K_A und der Component Manager `ready` signalisieren, beginnt die *Commit*-Phase (4).

b Im Fehlerfall oder bei Zeitüberschreitung – auch während des Commit – wird die *Abort*-Phase (5) eingeleitet.

4 Commit Für den Austausch werden die folgenden Schritte abgearbeitet:

1. Handelt es sich bei K_A um eine sichtbare UI-Komponente, wird sie durch den Aufruf ihrer Methode hide versteckt ($S_A = Active$).
2. Danach wird K_A durch den Aufruf von disable deaktiviert ($S_A = Inactive$).

3. Der Component Manager ruft nun die Komponentenbeschreibung von K_B vom Integration Service ab und liest die enthaltenen Informationen ein.
4. Für K_B wird der Wrapper W_B instanziiert und gemäß den Informationen aus MatchingResult und Binding konfiguriert. Auch er ist initial blockiert und leitet somit keine Nachrichten an K_B weiter.
5. K_B wird instanziiert ($S_B = Instantiated$)
6. Alle Verweise auf W_A, z. B. von angeschlossenen Links im Event Broker, werden auf W_B gesetzt ($S_B = Integrated$) und die Liste vorgehaltener Nachrichten von W_A auf W_B übertragen.
7. K_B wird initialisiert ($S_B = Active$). Die vorherige Integration in den Daten- und Kontrollfluss ermöglicht anderen Modulen die Reaktion auf initiale Events von K_B.
8. Es erfolgt der Zustandstransfer von K_A zu K_B, wie in Abschnitt 6.3.3.3 beschrieben wird.
9. Falls K_A zuvor sichtbar war, wird K_B durch den Aufruf ihrer Methode show sichtbar gemacht ($S_B = Interactive$).
10. Die Methode dispose von K_A wird ausgeführt, was die Freigabe all ihrer komponenteninternen Ressourcen nach sich zieht.
11. Der Destruktor von K_A wird aufgerufen, der dessen Binding entnommen werden kann.
12. W_A wird entfernt.

5 Abort Fehler während der Prepare- oder Commit-Phase müssen durch den Component Manager behandelt werden. Die bereits im Rahmen der Integration (vgl. Abschnitt 6.2.3.2) diskutierten Fehlerfälle gelten auch hier, mit einigen Erweiterungen:

Sicherungsfehler Erreicht K_A keinen sicheren Zustand, d. h. die Prepare-Phase schlägt fehl, wird die Blockierung aufgehoben und der Austausch abgebrochen.

Ladefehler Kann K_B nicht korrekt geladen werden, z. B. aufgrund von Verbindungsproblemen, wird K_A wieder aktiviert und ggf. eingeblendet.

Instanziierungsfehler Falls K_B nicht instanziiert werden kann, wird K_A ebenfalls aktiviert und W_B kann entfernt werden.

Integrationsfehler Verläuft die Integration fehlerhaft, müssen alle Verweise zurück auf W_A gesetzt werden und der Abbruch wie oben erfolgen.

Initialisierungsfehler Wenn K_B nicht initialisiert werden kann, wird wie beim Integrationsfehler vorgegangen.

5. Unblock Schließlich werden die Adaptions- und Kommunikationssperren gelöst.

1. Die initiale Blockierung von W_B wird aufgehoben, wodurch zwischengespeicherte Nachrichten an K_B weitergeleitet werden.
2. Schließlich wird die Adaptionssperre des Adaptation Managers gelöst, sodass ausstehende Advices zur Anwendung kommen können.

Mit dem erfolgreichen Abschluss dieser Schritte ist der Komponententausch vollzogen und K_B besitzt den gleichen öffentlichen Zustand wie vormals K_A. Es ist zu erkennen, dass sich der Algorithmus prinzipiell auf den Austausch einer Komponente durch eine andere konzentriert. Ein $m : n$-Austausch würde eine Erweiterung des Discovery-Prozesses sowie die Unterstützung von Komponentenmengen nötig machen. Dies ist konzeptionell beherrschbar, geht aber über den Rahmen dieser Arbeit hinaus.

6.3.3.3 Zustandstransfer

Im oben beschriebenen Ablauf wurde in der Commit-Phase auf den Zustandstransfer verwiesen. In diesem Zusammenhang muss man sich den Zustandsbegriff ins Gedächtnis rufen, der in Abschnitt 5.1.1 für Mashup-Komponenten eingeführt wurde. Der Zustand ist demnach durch die Gesamtheit der Properties einer Komponente bzw. ihrer Belegung charakterisiert. Andere Informationen können und sollen der Kompositionsumgebung – auch aufgrund der Unabhängigkeit von der Komponentenimplementierung – nicht zur Verfügung stehen.
Der Zustandstransfer folgt dem von Gamma et al. (1995) beschriebenen Memento-Muster, welches das Zusammenspiel von *Originator* (Komponenten) und *Caretaker* (Adaptation Manager) zur Sicherung und Wiederherstellung von Zuständen beschreibt. Beim Wechsel zwischen zwei Komponenten K_A und K_B müssen zunächst die Belegungen aller Properties von K_A ausgelesen werden, um sie danach in K_B zu setzen. Dazu kommen die in Abschnitt 5.1.3.2 angesprochenen, impliziten Getter- und Setter-Methoden zum Einsatz.
Durch das optionale Attribut `transient` können Properties in der MCDL als flüchtig gekennzeichnet werden. Dadurch gibt der Komponentenentwickler an, dass sie im Fall des Komponententauschs nicht gesichert werden müssen, z. B. falls sie sich implizit durch interne Berechnungen ergeben.

Semantische Inkompatibilitäten sind beim Tausch ausgeschlossen, da beide Komponenten durch die vorgestellten Discovery-Algorithmen der gleichen semantischen Vorlage entsprechen müssen. Zwei Konstellationen syntaktischer Differenzen müssen jedoch beachtet werden:
Einerseits können trotz semantisch kompatibler Datentypen unterschiedliche syntaktische Repräsentation in K_A und K_B verwendet werden, wenn eine Komponente durch einen Aspekt mit einer Alternative ersetzt wird. Vor dem Setzen dieser Eigenschaften in K_B müssen deshalb die in Abschnitt 6.2.4.2 beschriebenen Mediationsmechanismen zum Einsatz kommen.
Weiterhin besteht die Möglichkeit, dass K_A mehr Properties besitzt, als die Vorlage verlangt. Folglich können nicht alle Eigenschaften auf K_B übertragen werden. Dies ist prinzipiell unproblematisch, da alle prinzipiell nötigen Properties im Modell repräsentiert sein sollten und die restlichen ignoriert werden können. Allerdings kann dieses Vorgehen langfristig zum Datenverlust führen, z. B. beim späteren Wechsel zurück zu K_A. Unter den gegebenen Randbedingungen lässt sich das Problem nicht gänzlich lösen, da eine Zwischenspeicherung der überschüssigen Eigenschaften von K_A durch die Laufzeitumgebung bis zu einem etwaigen späteren Einsatz zu Inkonsistenzen führen kann, falls sich der Zustand zwischenzeitlich (bei K_B) ändert.

Die folgenden Punkte skizzieren den Austausch von K_A durch K_B:

1. Zuerst werden alle zustandsgebenden, d. h. nicht-transienten Eigenschaften von K_A ermittelt. Dazu wird ihre MCDL analysiert, die entweder lokal gehalten oder vom CoRe abgerufen wird.
2. Alle ermittelten Eigenschaften von K_A werden durch Aufruf der Methode getProperty ausgelesen. Die Abfrage über den Wrapper der Komponente stellt sicher, dass syntaktische Differenzen zwischen Properties von K_A und dem zugrunde liegendem Template im Kompositionsmodell durch die Mediationsverfahren aufgelöst werden. Es folgt die Sicherung der ausgelesenen Eigenschaften.
3. Der Zustand von K_B wird durch den Aufruf der Methode setProperty für alle gesicherten Properties gesetzt. Transienten Eigenschaften werden die Initialbelegungen aus dem Kompositionsmodell zugewiesen. Sind diese nicht definiert, werden die Standardwerte aus der MCDL verwendet. Überzählige Properties, die nicht auf K_B abgebildet werden können, werden verworfen. Auch hier erfolgt der Zugriff wieder über den Wrapper.

Die vorgestellten Konzepte erlauben den dynamischen Austausch von Komponenten zur Laufzeit. Durch das Proxy-Konzept werden die Isolation der Aktion und die Schnittstellenkompatibilität der beteiligten Komponenten hergestellt. Die Atomarität und Konsistenz der Aktion sichert ein Phasen-Commit-Protokoll, welches im Fehlerfall einen *Rollback* erlaubt. Als Teil des Austauschprozesses wurde der Zustandstransfer erläutert, der dafür sorgt, dass alle zustandsgebenden Eigenschaften auf die jeweils neue, semantisch kompatible Komponente übertragen werden.

Diese Betrachtungen schließen die Diskussion der Adaptionskonzepte aus der Sicht der Laufzeitumgebung ab. Der folgende Abschnitt fasst die Inhalte und Konzepte dieses Kapitels zusammen und verdeutlicht die wissenschaftlichen Beiträge im Hinblick auf die in Abschnitt 1.1.3 formulierten Ziele.

6.4 Zusammenfassung und Diskussion

Ausgehend von den im letzten Kapitel eingeführten Konzepten zur Modellierung wurden in diesem Kapitel die Laufzeitaspekte der universellen Komposition beleuchtet. Schwerpunkte bildeten die wissenschaftlichen Konzepte zur kontextsensitiven Suche, Auswahl und Integration von Mashup-Komponenten, zur koordinierten Ausführung der Komposition sowie zur dynamischen, kontextabhängigen Anpassung der Kompositionen zur Laufzeit.
Die folgenden Abschnitte fassen die Alleinstellungsmerkmale und wissenschaftlichen Beiträge der vorgestellten Lösungen zusammen und nehmen dazu Bezug auf die in Abschnitt 2.3 formulierten Anforderungen, die jeweils mit ✪ hervorgehoben sind.

Kontextsensitiver Integrationsprozess

In Teilkapitel 6.1 wurden Konzepte zur ✪ Discovery vorgestellt, die die kontextsensitive Suche von Mashup-Komponenten unter funktionalen und nicht-funktionalen Gesichtspunkten und unter Beachtung von Kontexteigenschaften zur Laufzeit erlauben. Grundlage hierfür bieten die im letzten Kapitel vorgestellten Konzepte zur ✪ Abstraktion im Kompositionsmodell.
Die Suche und Auswahl der Komponenten erfolgt auf Basis semantischer Vorlagen in Anlehnung an die Phasen des SWS Nutzungsprozesses (Abschnitt 3.1.2). Das

vorgestellte, logikbasierte Suchverfahren nutzt die Komponenten- und Kompositionsbeschreibung, um Subsumptionsbeziehungen zwischen Anfragen (Vorlagen) und vorhandenen Komponenten auf Basis derer ✪ Semantik auszuwerten. Dabei können Teiltreffer unterstützt und syntaktische Abweichungen bis zu einem gewissen Grad im Rahmen der ✪ Mediation überbrückt werden. Beides fördert die ✪ Interoperabilität und ✪ Wiederverwendbarkeit von Komponenten.

Die Einbeziehung nicht-funktionaler Semantik erlaubt den Abgleich mit Kontextparametern und somit die ✪ Kontextualisierung bereits im Auswahlprozess. Neben kontextspezifischen Ausschlusskriterien kann auch die Wichtung von Kandidaten durch Kontext- und sonstige Metadaten beeinflusst werden. Die entsprechenden Ranking-Regeln sind als Teil eines Kompositionsmodells, aber auch unabhängig formulierbar. Ihre ✪ Plattformunabhängigkeit erlaubt die Nutzung in verschiedenen Anwendungen. Bei der Adressierung semantischer Daten kommt generell SPARQL zum Einsatz, was zur ✪ Standardkonformität und Güte der Lösung beiträgt.

Grundlage für die Auswahl der passendsten Komponente bildet schließlich die Gesamtbewertung aus funktionalem Deckungsgrad und kontextspezifischer Bewertung. Ihre ✪ späte Bindung erfolgt durch die MRE zur Laufzeit auf Basis der Informationen aus dem *Binding* der Komponentenbeschreibung und o. g. Abbildungsvorschriften.

Nicht zuletzt aufgrund des universellen, zustandsbehafteten Komponentenmodells gehen die vorgestellten Konzepte über die Lösungen der SWS hinaus. Im Gegensatz zu vergleichbaren Kompositionsansätzen aus dem SOA- und CBSE-Bereich werden erstmals Komponenten der Präsentationsebene unterstützt. Dabei wird kein generativer Ansatz, wie bei ServFace genutzt, sondern Komponenten direkt in die Anwendung integriert. Es besteht jedoch keine Beschränkung auf die UI, wie beim SOAUI-Konzept. Vergleichbare Ansätze im Mashup-Umfeld existieren nicht.

Kompositionsinfrastruktur und Laufzeitumgebung

Den Schwerpunkt von Teilkapitel 6.2 bildete die Vorstellung einer serviceorientierten Referenzarchitektur zur universellen Komposition. Sie realisiert alle Belange von der Modellverwaltung bis zur Umsetzung des Integrationsprozesses und der Ausführung kompositer Anwendungen. Entsprechend der ✪ SoC werden die Verantwortlichkeiten bei der Komposition durch dedizierte Dienste erbracht.

Als erste Vertreter wurden die Dienste der ✪ Modellverwaltung vorgestellt. So erlaubt das CoRe die anwendungs- und plattformunabhängige Verwaltung von Mashup-Komponenten und entlastet die Laufzeitumgebung von der aufwändigen Validierung und Konsistenzsicherung registrierter Beschreibungen. Aufgrund der zentralen Verfügbarkeit und ständigen Sicherung kann Fehlern bei der Integration vorgebeugt werden, wodurch ✪ Performanz und Qualität der resultierenden Anwendungen steigen. Dem TyRe kommt die Verwaltung der semantischen Domänenmodelle zu, die mit Grounding-Definitionen sowie Liftings und Lowerings angereichert sind. Sie bilden die Grundlage für die ✪ Mediation.

Im Weiteren lag der Fokus der Betrachtung auf der Ausführungsumgebung – der MRE. Es setzt sich aus einer Reihe von Modulen zusammen, deren technologische Umsetzung und Verteilung zwischen Client und Server je nach Plattform variieren können. Zur Realisierung der Belange stützen sich alle Module auf das im letzten Kapitel vorgestellte Kompositionsmodell. Sie ermöglichen somit die modellbasierte

Entwicklung im Sinne der ✪ Plattformunabhängigkeit. Neben den modellierten Belangen decken sie insbesondere die dynamische Integration bzw. ✪ späte Bindung von Komponenten ab, wozu sie auf die Dienste des o. g. Integrationsprozesses zugreifen. Auch die Zustandsverwaltung und -sicherung, das ✪ Life-Cycle-Management der Komponenten sowie die plattformspezifische Umsetzung von Layouts und Sichten gehören zu den Aufgaben der MRE.

Zur Realisierung des Daten- und Kontrollflusses stellt die MRE den Event Broker bereit. Er setzt die Validierung und Vermittlung von Ereignissen den enthaltenen Nachrichten entsprechend der modellierten Links auf Basis einer Publish/Subscribe-Architektur um, was die größtmögliche ✪ lose Kopplung der dynamisch integrierten Bestandteile erlaubt. Die Mächtigkeit der Konzepte zur ✪ Koordination geht über die unidirektionale Verknüpfung von Ein- und Ausgängen, wie sie in existierenden Lösungen zur Dienst- und Mashup-Komposition vorherrscht, hinaus. Es werden insbesondere erweiterte Muster, z. B. Request-Response-Beziehungen und permanente, asynchrone Aktualisierungen sowie Synchronisation von Komponenten unterstützt. Weiterhin wurde anhand von Drag-and-Drop gezeigt, wie implementierungs- und endgerätespezifische Interaktionstechniken auf generische Schnittstellen der MRE abgebildet werden können. Vergleichbare Konzepte finden sich bislang in keiner der betrachteten Mashup-Plattformen.

Zur Steigerung der ✪ Interoperabilität wurden verschiedene Konzepte der ✪ Mediation vorgestellt, die die Überbrückung syntaktischer Differenzen zwischen Kommunikationspartnern sowie zwischen integrierten Komponenten und ihren Vorlagen erlauben. Die Modellkonformität auf der Signaturebene, z. B. hinsichtlich der Benennung, Anzahl und Reihenfolge von Parametern eines Events, wird durch Wrapper sichergestellt, die Komponenten kapseln und als Adapter fungieren. Die ✪ Mediation von Daten erfolgt auf dem Kommunikationsweg über den Umweg der semantischen Repräsentation. Das Vorgehen ist angelehnt an die Konzepte der SWS, bildet jedoch eine eigenständige Lösung, die explizit auf das universelle Komponentenmodell und die erweiterten Kommunikationsmuster ausgerichtet ist. Vergleichbare automatische Mediationsmechanismen existieren für keine der bekannten Web- und Mashup-Engineering-Ansätze.

Für den Zugriff auf externe Dienste und Ressourcen bietet die MRE eine einheitliche Zugriffsschicht, die von Plattformeigenschaften, Diensttypen und Protokollen abstrahiert. Neben der Vereinfachung für Komponentenentwickler ist für die MRE die Möglichkeit gegeben, die Same Origin Policy (SOP) zu umgehen und den Datenverkehr validieren und ggf. beeinflussen zu können.

Kontextverwaltungs- und Adaptionskonzepte

In Teilkapitel 6.3 wurden schließlich Konzepte zur dynamischen Adaption von kompositen Mashup-Anwendungen vorgestellt. Kern der Betrachtung bildete ein Adaptionssystem, welches im Zusammenspiel mit einer MRE die Interpretation der in Abschnitt 5.3.2.2 eingeführten Adaptionsaspekte erlaubt.

Die ✪ Kontextverwaltung wird durch den Kontextdienst CroCo abgedeckt, welcher aufgrund seiner ✪ Plattformunabhängigkeit durch verschiedene MREs genutzt werden kann. Somit bleiben die aufwändigen Belange der Kontextmodellierung und -verarbeitung samt Konsistenzsicherung und Schlussfolgerung neuen Wissens entkoppelt von der Ausführung kompositer Anwendungen. Durch die ✪ Erweiterbarkeit

von CroCo mit Profilen ist trotzdem sichergestellt, dass alle von Anwendungen benötigten Konzepte im Kontextmodell repräsentiert sind. Die Nutzung semantischer Modelle zur Kontextrepräsentation erlaubt indes die indirekte Verknüpfung von Komponenten-, Kompositions- und Kontextmodell über semantische Konzepte.
Die ✪ Kontextualisierung kompositer Anwendungen erfolgt durch die Auswertung der in Abschnitt 5.3 beschriebenen Modellkonstrukte. *Context Links* werden direkt auf die Publish-Subscribe-Infrastruktur abgebildet, während die Adaptionsaspekte in Form von ECA-Regeln repräsentiert und durch Ereignisse ausgelöst werden. Das Adaptionssystem – prinzipiell ein komplexes Modul der MRE – wertet dann zunächst die formulierten Bedingungen aus. Dazu greift es u. a. auf eine lokale Teilrepräsentation des Kontextmodells zurück, die ständig mit dem Kontextdienst synchronisiert wird. Netzwerklast und Anfragezeiten werden so möglichst gering gehalten. Sind alle Bedingungen erfüllt, erfolgt die Anpassung durch plattformspezifische Implementierungen der Adaptionsaktionen.
Aufgrund der ✪ Universalität des Komponentenmodells können sie sich auf alle Anwendungsebenen beziehen und somit inhaltlicher, funktionaler oder visueller Natur sein. In ihrer Mächtigkeit gehen die Adaptionstechniken deshalb über die Ansätze der Adaptive-Hypermedia hinaus, die sich i. d. R. auf Hypermedia-Konzepte wie Seiten, Links und *Closed-Corpus*-Inhalte beschränken. Gegenüber Lösungen aus dem CBSE- und SOA-Umfeld stellt hingegen die Anpassung zustandsbehafteter und interaktiven Komponenten ein Novum dar.
Dies macht eine gesonderte Behandlung beim Komponententausch nötig, um u. a. auf der Benutzeroberfläche Konsistenz zu bewahren, indem der Zustand zwischen alter und neuer Instanz transferiert wird. Zu diesem Zweck wurde ein Phasen-Commit-Protokoll vorgestellt, welches den Austausch und ✪ Zustandstransfer unter Sicherstellung der Konsistenz der Anwendung erlaubt. Das bereits mehrfach erwähnte Wrapper-Konzept dient hierbei (und auch bei anderen Adaptionsaktionen) der Isolation und verhindert den zwischenzeitlichen Datenverlust.

Die in diesem Kapitel vorgestellten Konzepte der Kompositions- und Laufzeitumgebung vervollständigen die Vision der modellgetriebenen Entwicklung kompositer Anwendungen unter Berücksichtigung aller Anforderungen aus Abschnitt 2.3. Die Umsetzbarkeit und Praktikabilität der Konzepte muss jedoch noch bewertet werden. Das nächste Kapitel gibt deshalb Auskunft darüber, in welcher Form die beschriebenen Modellierungs- und Laufzeitkonzepte umgesetzt wurden, und veranschaulicht ihre Praktikabilität anhand einiger Beispielanwendungen.

7
Umsetzung und Validierung der Konzepte

Zu Beginn dieser Arbeit wurden die Herausforderungen und Probleme bei der Entwicklung moderner, serviceorientierter Webanwendungen identifiziert und diesbezüglich eine Reihe von Forschungszielen formuliert. Anhand von drei Anwendungsszenarien wurden in Kapitel 2 konkrete Anforderungen an ihre Umsetzung abgeleitet, vor deren Hintergrund in Kapitel 3 zunächst der Stand der Forschung und Technik untersucht und bewertet wurde. Nach einem Überblick über die Gesamtlösung wurden in den letzten beiden Kapiteln 5 und 6 die Teilkonzepte zur plattformunabhängigen Modellierung, dynamischen Integration und adaptiven Ausführung kompositer Anwendungen vorgestellt. Dieses Kapitel erläutert nun die technische Realisierung und Validierung der entwickelten Konzepte, die vorrangig im Rahmen des Forschungsprojektes CRUISe [@CRUISe, 2012] entstanden.

Der Schwerpunkt in Abschnitt 7.1 liegt zunächst auf der Umsetzung der Modellierungskonzepte zur universellen Repräsentation, Be- und Verarbeitung von Komponenten und Kompositionen. So werden u. a. die Formalisierung von Komponentenbeschreibung und Kompositionsmodell und die prototypische Umsetzung der Unterstützungsmechanismen im Autorenprozess vorgestellt.
Abschnitt 7.2 widmet sich danach der Realisierung der Laufzeitkonzepte. Diese umfassen zum einen die Bestandteile der verteilten Kompositionsinfrastruktur und die Realisierung des in Abschnitt 6.1 konzipierten Integrationsprozesses. Zum anderen werden verschiedene Implementierungen der MRE vorgestellt und am Beispiel der clientseitigen Laufzeitumgebung die Umsetzung der Konzepte zur Kontextualisierung bzw. Adaption von Kompositionen zur Laufzeit erläutert.
Schließlich widmet sich Abschnitt 7.3 den umgesetzten Beispielanwendungen, die sich an den in Abschnitt 2.2 skizzierten Szenarien ausrichten und veranschaulichen, inwiefern die geschaffenen Konzepte in der Praxis den identifizierten Problemen und Herausforderungen gerecht werden.

7.1 Realisierung der Modellierungsmittel

In Kapitel 5 wurden im Zusammenhang mit dem universellen Komponentenmodell verschiedene Beschreibungsformen vorgestellt, die die abstrakte, deklarative Spezifikation von Mashup-Komponenten in verschiedener Mächtigkeit erlauben. Weiterhin wurde ein Modell konzipiert, welches die plattformunabhängige, abstrakte Repräsentation interaktiver Anwendung als Komposition derartiger Komponenten erlaubt. Die folgenden Abschnitte widmen sich der Umsetzung und Validierung dieser Konzepte. Dazu wird zunächst auf die Formalisierung der Komponentenbeschreibung durch XML-Schemata und Ontologien eingegangen. Danach steht die Realisierung des Kompositionsmodells samt zugehöriger Autorenwerkzeuge und Unterstützungsmechanismen im Vordergrund.

7.1.1 Komponentenbeschreibung in XML und OWL

Zur Beschreibung von Komponenten wurde das in Abschnitt 5.1.3 vorgestellte Vokabular mittels XML-Schema (zur Spezifikation der MCDL und SMCDL) und der Web Ontology Language (OWL) (im Falle der MCDO) modelliert.
Das Schema der MCDL-Beschreibung wurde bereits in Abbildung 5.4 auszugsweise veranschaulicht und seine Anwendung anhand von Codebeispiel 5.1 illustriert. Die SMCDL bildet eine Erweiterung des MCDL-Schemas, die im Wesentlichen zusätzliche Elemente und Attribute zur semantischen Annotation und Typisierung hinzufügt. Ein Beispiel einer solchen Beschreibung findet sich in Anhang A.1.
Die Umsetzung der MCDO erfolgte auf Basis von OWL DL. Abbildung 7.1 zeigt die Struktur der Ontologie anhand der wichtigsten Klassen und Relationen. Die bereits bekannten Elemente der SMCDL wurden dazu i. d. R. als Ontologieklassen repräsentiert. So ist in der Abbildung die Aufteilung einer MashupComponent in MetaData, Interface und Binding zu erkennen. Letzteres enthält Klassen, die Abhängigkeiten, Instanziierungs- und Zugriffsinformationen enthalten, welche u. a. über CodeTemplates ausgedrückt werden können. Zur Formalisierung bestimmter Teilaspekte, z. B. nicht-funktionaler Eigenschaften als Teil der Metadaten, wurden bestehende Domänenmodelle bzw. Ontologien integriert.
Auch die wesentlichen Bestandteile der öffentlichen Schnittstelle (Interface) sind erkennbar. Gegenüber der XML-Repräsentation enthält die MCDO jedoch auch Erweiterungen, wie z. B. Vorbedingungen für Operationen, die durch die Klasse Precondition ausgedrückt werden. Die Klasse Capability – hier in Bezug auf ein Ereignis eingeblendet – bildet die funktionalen, semantischen Annotationen im Modell ab (vgl. XML-Attribut `functionality` der MCDL).
Alle entwickelten Beschreibungsformate wurden in einem Eclipse-Projekt zusammengefasst (de.tudresden.inf.cruise.models[1]). Neben den Schemata enthält es eine Java-API zur Verarbeitung von Beschreibungsinstanzen, die für SMCDL mit der bewährten Referenzimplementierung der Java Architecture for XML Binding (JAXB) [@Oracle, 2012] umgesetzt wurde. Zur Verarbeitung aller Ressourcen der MCDO wurde analog eine API über *schemagen* des Jena Frameworks [@Jena, 2011] erstellt. Beide kommen u. a. im CoRe zum Einsatz und stehen gleichermaßen Autorenwerkzeugen und serverseitigen MREs zur Verfügung.

[1] Der Namensraum de.tudresden.inf.cruise wird in den folgenden Abschnitten mit ⋇ verkürzt.

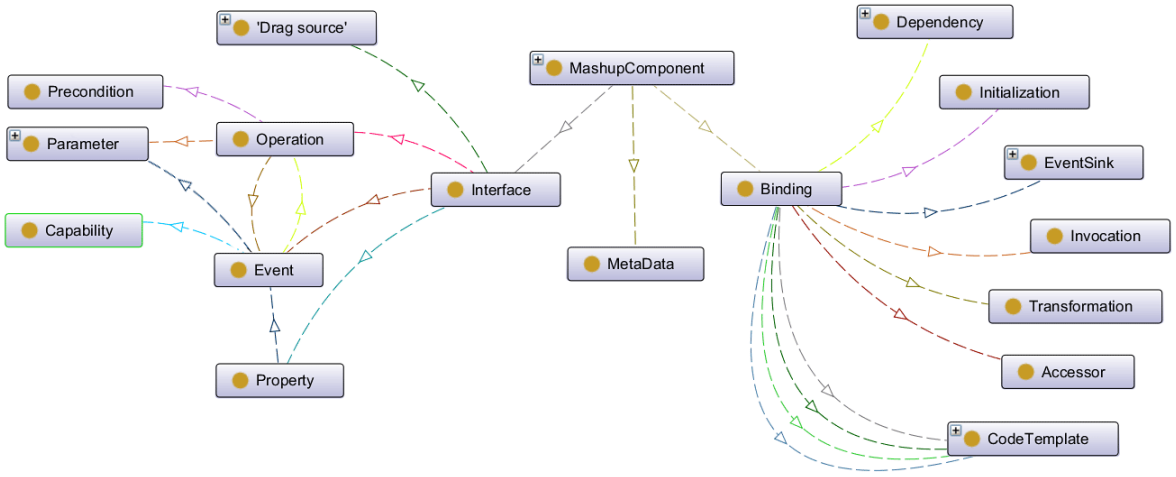

Abb. 7.1: Klassen und Relationen der MCDO im Überblick (Auszug)

7.1.2 EMF-basiertes Kompositionsmodell

Wie bereits in Abschnitt 5.2 einleitend beschrieben, ist das vorgestellte Kompositionsmetamodel auf der M2-Ebene der MOF-Architektur angesiedelt. Es befindet sich somit auf der gleichen Ebene wie die UML, was zunächst die Modellierung über dedizierte UML-Profile, wie z. B. in UWE, nahelegt. Bei genauerer Betrachtung ist dies jedoch problematisch: Zum einen führt die Bindung an die Metamodellklassen der UML zu Beschränkungen, denn nicht alle geforderten Assoziationen und Vererbungshierarchien lassen sich darauf abbilden. Zum anderen würde die Erweiterung zu einer Verfälschung der Metaklassen führen. Deshalb wurde von der Nutzung von UML-Profilen Abstand genommen.
Das konzipierte Kompositionsmodell wurde vielmehr als eigenständiges Metamodell umgesetzt, welches genau auf die in Abschnitt 5.2 vorgestellten Eigenschaften abgestimmt ist (vgl. Reimann (2009)). Grundlage hierfür bot das Meta-Metamodell Ecore und somit das Eclipse Modeling Framework (EMF) [@Eclipse, 2011a]. Die Nutzung von Ecore erlaubt u. a. die Definition einer Komposition als MOF-konformes Modell und die Serialisierung als XML Metadata Interchange (XMI). Diese Standardkonformität kommt letztlich der Interoperabilität und Werkzeugunabhängigkeit bei der Modellierung zugute.
Codebeispiel 7.1 zeigt einen Ausschnitt einer solchen Modellserialisierung, an der die Aufteilung in die beschriebenen Teilmodelle (Zeilen 3, 15, 23, 27) deutlich wird. Der Auszug enthält die Definition einer Komponente zur Wetteranzeige (Zeile 5–12) und deren Platzierung in einem Layout (Zeile 18) und View (Zeile 25).

Bedingungen des Modells, die sich nicht über die Klassenbeziehungen ausdrücken lassen, wurden mit OCL [@OMG, 2010] formuliert und den jeweiligen Modellklassen als Annotationen hinzugefügt. Codebeispiel 7.2 zeigt eine solche Bedingung für die Klasse LayoutElement, die Blattknoten eines Layouts repräsentiert (vgl. auch Abbildung 5.18). Wenn dieses ein Unterlayout enthält, dann darf kein Verweis *location* auf eine UI-Komponente existieren (Zeile 2). Gleiches gilt in umgekehrter Reihenfolge, d. h. wenn kein Layout enthalten ist, darf der Verweis auf eine UI-Komponente nicht fehlen (Zeile 4). Die doppelte Negation im Beispiel ist dadurch bedingt, dass EMF lediglich die Funktion oclIsUndefined(), nicht jedoch oclIsDefined() unterstützt.

```
<?xml version="1.0" encoding="UTF-8"?>
<mcm:MashupComposition xmi:version="2.0" ... name="TravelMash">
 <conceptualModel>
  <components>
   <component xsi:type="mcm:UIComponent" name="Wetter" id="http://wetter.com/uis/1.8">
    <property type="http://.../travel.owl#Location" name="location"/>
    <operation name="update">
     <parameter type="http://.../travel.owl#Location" name="location"/>
     <functionality URI="http://.../travelPlanning.owl#WeatherInformation"/>
    </operation>
    ...
   </component>
   ...
  </conceptualModel>
  <layoutModel>
   <layout xsi:type="mcm:FillLayout" name="fixed">
    <bounds height="220" width="160" unit="pixel"/>
    <position locate="Wetter" x="0" y="180" unit="pixel">
      <bounds height="160" width="220" unit="pixel"/>
    </position>
    ...
   </layout>
  <layoutModel>
  <screenflowModel initialView="start">
   <view name="start" vLayout="fixed"/>
  </screenflowModel>
  <communicationModel>
   ...
  </communicationModel>
</mcm:MashupComposition>
```

Lst. 7.1: Serialisierung eines beispielhaften Kompositionsmodells in XMI

```
context LayoutElement inv :
  not self.layout.oclIsUndefined() implies self.locate.oclIsUndefined()
 and
  self.layout.oclIsUndefined() implies not self.locate.oclIsUndefined()
```

Lst. 7.2: *XOR*-Bedingung im LayoutModel auf Basis von OCL

Ein weiterer Vorteil der Nutzung von EMF besteht in der Möglichkeit, vollautomatisch eine Java-basierte API für das Metamodell erstellen zu lassen. Sie kann die Grundlage für Quellcodegeneratoren oder die Verarbeitung durch serverseitige Laufzeitumgebungen selbst darstellen. Alle Ressourcen des Metamodells wurden deshalb im Eclipse-Plug-in-Projekt ⁎.compositionmodel gekapselt, das die Modell-API über einen *Extension Point* weiteren Eclipse-Instanzen, z. B. Transformatoren und Editoren, zur Verfügung stellt.

Auf Basis der API kann mit den Mitteln von EMF ein Eclipse-Plug-in erzeugt werden, welches einen Baum-Editor zur Bearbeitung von Modellinstanzen enthält. Dieses Editor-Plug-in kann in bestehende Eclipse-Umgebungen eingebunden und erweitert werden. Zur Generierung von API und Editor wurde ein EMF *Generator Model* erstellt, welches neben den modellierten Klassen Informationen zur Code-Generierung enthält. Diese können beliebig angepasst bzw. konfiguriert werden, z. B. hinsichtlich der Benennung einzelner Klassen oder Pakete. Mit der „Anwendung“ des Generator Models erfolgte schließlich die Erstellung der Basis-API oder des erwähnten Editors in Form weiterer Plug-ins, auf die in Abschnitt 7.1.4 näher eingegangen wird.

7.1.3 Modelltransformationen

In Abschnitt 5.4 wurden die einzelnen Schritte bei der Modellierung kompositer Anwendungen, z. B. das Finden, Konfigurieren, Verknüpfen und Anordnen von Komponenten, umrissen und diesbezüglich Möglichkeiten der (semi-)automatischen Unterstützung von Entwicklern aufgezeigt. Die Umsetzung dieser Automatismen erfolgte durch die Definition von Modellverfeinerungen auf Basis der Transformationssprache Query View Transformation (QVT), die durch EMF bereitgestellt wird. Auch QVT ist Teil der MOF-Spezifikation, wodurch die Transformationen offen und einfach wiederverwendbar sind.
Alle Transformationen wurden mittels operationalem QVT (QVTO) realisiert und in einem eigenen Plug-in-Projekt ⁎.compositionmodel.m2m zusammengefasst. QVTO unterstützt die Einbindung sog. *Black Box Libraries*, die Abbildungslogik programmatisch beschreiben und in beliebigen Programmiersprachen umgesetzt sein können. In der prototypischen Implementierung kommt Java zum Einsatz.
Die realisierten Transformationen korrespondieren zu den in Abschnitt 5.4 vorgestellten Modellverfeinerungen. So können dem Conceptual Model durch die Angabe von Web-Service-Beschreibungen entsprechende Mashup-Komponenten hinzugefügt werden. Dazu wird u. a. auf die Bibliothek `WSDL4J` [@WSDL4J, 2012] zurückgegriffen, um lokale oder entfernte WSDL-Beschreibungen interpretieren und die darin spezifizierten Ein- und Ausgänge auf Operationen und Ereignisse von Service-Komponenten abbilden zu können.
Eine weitere Transformation dient der Generierung von Sichten aus dem Layout Model. Für jedes Master-Layout wird ein eigener View samt Transition erstellt. Letztere muss danach vom Entwickler nach seinen Vorstellungen konfiguriert werden.
Weiterhin wurde eine Transformation zur Generierung des initialen Communication Models umgesetzt. Hierbei erfolgt die Analyse aller Ereignisse und Operationen im Modell. Für jede Signatur wird ein Kanal (Link, BackLink) erstellt und mit den entsprechenden Artefakten (Publisher, Subscriber, usw.) verknüpft. Dem Entwickler obliegt dann ggf. die Anpassung an seine Bedürfnisse, z. B. das Anfügen neuer Teilnehmer unter Nutzung von ParameterMappings, die Definition von Aktualisierungsintervallen, etc.

Die Code-Generierung, z. B. in JavaServer Pages (JSP) für die serverseitige oder HTML und JavaScript für die clientseitige Laufzeitumgebung, erfolgt mittels Xpand [@Eclipse, 2012b]. Grundlage bieten Schablonen bzw. *Templates*, die verschachtelt aufgerufen werden und an entsprechend markierten Stellen Informationen aus dem Kompositionsmodell in den ausgegebenen Code integrieren. Ein Root-Template wird dazu auf das Wurzelelement des Modells angewendet und ruft in Folge weitere Templates für die spezifischen Teilmodelle auf. Deren Anwendung führt für die serverseitige MRE beispielsweise zur Generierung der grundlegenden JSP-Seite, CSS-Vorlagen und des benötigten JavaScript-Codes zur Initialisierung von Laufzeitumgebung und Anwendung. Diese Dateien werden schrittweise durch die weitere Templates gefüllt, bis lediglich Platzhalter für die zur Laufzeit zu bindenden Komponenten im Anwendungscode übrig bleiben.
Codebeispiel 7.3 gibt einen praktischen Einblick in die Nutzung von Xpand anhand des Templates zur Auswertung der Style-Informationen im Conceptual Model. Nach dem Import von Namensräumen und Skripten ist in Zeile 4 das eigentliche Template

zu sehen, umschlossen durch die Schlüsselworte «DEFINE» und «ENDDEFINE». Auf die Modellklasse Styles angewendet (Zeile 5) führt es für jeden enthaltenen Style die Anweisung «FILE» aus (Zeile 6), die zur Generierung jeweils einer CSS-Datei führt. Die Inhalte der Datei werden durch ein anderes Template createComponentStyle erstellt, welches durch «EXPAND» aufgerufen wird (Zeile 7).

```
1  «IMPORT compositionmodel»
2  «EXTENSION extensions::utilities»
3
4  «DEFINE createComponentStyles(String prefix) FOR Styles»
5    «FOREACH this.style AS currentStyle ITERATOR styles-»
6      «FILE "styles/component_" + currentStyle.decorates.id + ".css"-»
7          «EXPAND createComponentStyle(prefix+styles.counter1) FOR currentStyle-»
8      «ENDFILE-»
9    «ENDFOREACH-»
10 «ENDDEFINE»
```

Lst. 7.3: Xpand-Template zur Generierung von *Styles*

Durch den hierarchischen und polymorphen Template-Aufruf wird schließlich die gesamte Anwendung erstellt. Zur Modell-Validierung kommt in diesem Zusammenhang die Sprache *Check* zum Einsatz, die im Xpand-Framework enthalten ist. Alle benötigten Ressourcen – in erster Linie die Xpand Templates – sind, je nach Zielplattform, in entsprechende Plug-ins gekapselt, z. B. ⁎.compositionmodel.m2c.tsr für die Generierung clientseitigen Codes.

7.1.4 Modellierungswerkzeug

Die Entwicklung von Autorenwerkzeugen stand nicht im Fokus der Arbeit, wurde aber zur Erleichterung von Test und Evaluation nötig und deshalb parallel betrieben. Grundlage hierfür bot der bereits angesprochene, reflexive Modelleditor aus dem EMF-Framework, der um einige Plug-ins erweitert wurde. Neben der im letzten Abschnitt vorgestellten Transformationslogik ermöglichen diese u. a. die Anbindung des CoRe, um dort registrierte Komponenten direkt einbinden zu können.
Der resultierende *Mashup Composition Editor* ist in Abbildung 7.2 zu sehen. Er ist in zwei Bereiche geteilt: Im Hauptbereich (A und D zeigen verschiedene Teile des Modells) findet sich eine hierarchische Ansicht des Kompositionsmodells, die das Hinzufügen und Entfernen neuer Knoten bzw. Instanzen erlaubt. Rechts daneben (B, C) sind verschiedene Ansichten zur Konfiguration der einzelnen Klassen zu sehen.
Der *Component Browser* (B) ist eine Sicht, die zur Anzeige von im CoRe registrierten Komponenten dient. Alternativ können sie aus einem lokalen Ordner geladen werden. Die Browser-Ansicht erlaubt es, Komponenten nach ihrer Art zu sortieren und sich ihre Beschreibung anzusehen. Der eigentliche Nutzen besteht darin, sie per Doppelklick in das Modell zu übernehmen. Im Hintergrund wird die jeweils zugehörige SMCDL-Beschreibung vom CoRe abgerufen, eingelesen und auf neue Instanzen der Kompositionsmodellklassen abgebildet. Diese werden dem *Conceptual Model* hinzugefügt und sind somit direkt Teil der kompositen Anwendung. Der Aufwand der manuellen Modellierung von Komponenten entfällt.
Der *Property View* (C) gibt Auskunft über die im Baum ausgewählten Entitäten des Modells. Hier können ihre Eigenschaften gesetzt bzw. geändert werden – in der Abbildung betrifft dies die Attribute der selektierten UI-Komponente.

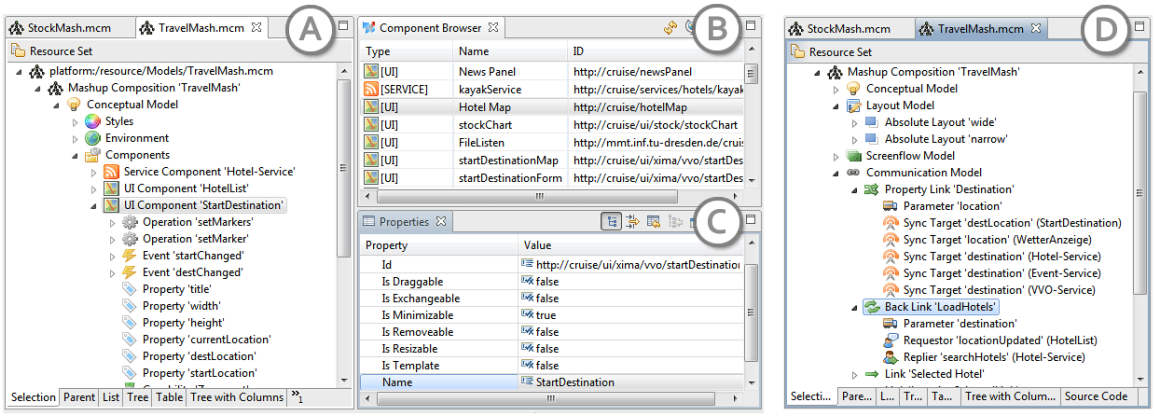

Abb. 7.2: Der *Mashup Composition Editor* in Aktion

Die Validierung abgeschlossener Kompositionen umfasst neben der Analyse der Modellkonformität auch die Überprüfung der inhärenten OCL-Constraints. Diese erfolgt auf Basis von Mechanismen, die durch EMF bereitgestellt werden. Die Validität der serialisierten Modelle ist somit sichergestellt.

Für das Deployment aus dem Editor heraus wurden verschiedene Varianten geschaffen. Im einfachsten Fall erfolgt die Serialisierung des Modells in XMI. Sie wird durch entsprechende MREs, wie die in Abschnitt 7.2.2.2 vorgestellte, serverseitige Plattform, interpretiert.
Eine weitere Variante besteht in der statische Umwandlung des Kompositionsmodells in eine ausführbare Anwendung auf Basis der im letzten Abschnitt erwähnten Transformationenbeschreibungen in Xpand. Dies wurde prototypisch für eine JSP-basierte Laufzeitumgebung sowie eine frühe Version der clientseitigen TSR aus Abschnitt 7.2.2.1 realisiert.
Für die späteren Ausbaustufen der TSR, die die direkte Modellinterpretation unterstützen, wurde ein weiteres Plug-in geschaffen. Es dient der Kapselung von Modell und vorkonfigurierter Ausführungsumgebung als fertig ausführbare, clientseitige Webanwendungen. Nach der strukturellen und OCL-basierten Validierung werden dazu alle nötigen Dateien und Bibliotheken der clientseitigen Laufzeitumgebung geladen und eine XHTML-Datei zu deren Initialisierung generiert. Diese enthält Anweisungen zur Instanziierung der MRE-Module sowie Konfigurationsparametern, wie den Adressen, unter denen Kompositionsmodell, Komponenten-Repository, Kontextdienst usw. zu erreichen sind. Zuletzt wird die Anwendung zu einem Web Application Ressource (WAR)-Archiv gepackt und kann aus dem Editor heraus auf einem vom Autor spezifizierten Servlet-Container deployt werden.

Die Konzepte aus Kapitel 5 konnten somit vollständig umgesetzt werden. Die Modellierung von Komponenten und Anwendungen, wie später in Abschnitt 7.3 beschrieben, erwies sich sowohl als praktikabel als auch ausreichend mächtig, machte jedoch die Notwendigkeit guter Autorenwerkzeuge deutlich. Im Hinblick auf die Zielgruppe der Nicht-Programmierer bedarf es deshalb weiterer Abstraktionen, z. B. in Form visueller Editoren, die direkt auf den geschaffenen Modellschnittstellen aufsetzen können. Vor der Diskussion der Modellierung am Beispiel, widmet sich der folgende Abschnitt zunächst der Realisierung der Laufzeitaspekte.

7.2 Realisierung der Kompositions- und Laufzeitumgebung

Zur Umsetzung und Validierung der in Kapitel 6 vorgestellten Konzepte wurden die Bestandteile der verteilten Kompositionsinfrastruktur sowie verschiedene Laufzeitumgebungen für die kompositen Anwendungen realisiert. Die folgenden Abschnitte geben einen Überblick über die Ergebnisse, wobei auch Verteilungsaspekte diskutiert werden. Zunächst widmet sich Abschnitt 7.2.1 der Realisierung von Komponenten- und Modellverwaltung sowie der Discovery-Algorithmen. In Abschnitt 7.2.2 bildet die Darstellung verschiedener Implementierungen der MRE den Schwerpunkt, die die clientseitige und serverseitige Komposition sowie die Integration mit Geschäftsprozessen ermöglichen. Schließlich wird in Abschnitt 7.2.3 gezeigt, wie die Kontextverwaltungs- und Adaptionskonzepte umgesetzt und validiert wurden.

7.2.1 Semantische Verwaltung und Discovery

Ausgangspunkt jeder Komposition bildet die Menge der Mashup-Komponenten, deren Verfügbarkeit sowohl zur Entwicklungs- als auch zur Laufzeit sichergestellt sein muss. Darauf aufbauend bilden die Algorithmen zur funktionalen und nichtfunktionalen Suche bzw. Auswahl von Komponenten ein zentrales Konzept dieser Arbeit. Auf die Umsetzung beider Aspekte wird im Folgenden eingegangen.

7.2.1.1 Komponenten- und Typverwaltung

Die permanente Verfügbarkeit qualitativ hochwertiger Komponenten bildet die Grundlage jeder Komposition. Je größer die Komponentenauswahl ist, umso höher ist die Chance, eine passgenaue Implementierung zu finden. Insofern kommt der Verwaltung und Konsistenzsicherung der Komponentenmenge eine wichtige Rolle in der Infrastruktur zu, der mit der Umsetzung des CoRe Rechnung getragen wurde.

Komponentenverwaltung im CoRe

Zur semantischen Verwaltung und Persistierung von Komponentenbeschreibungen wurde das CoRe in Java implementiert. Für die Bereitstellung als Web Service kommt Apache Axis2 [@Apache, 2011] zum Einsatz. Auf dessen Basis wird eine SOAP-Schnittstelle für die Registrierung und Aktualisierung, den Abruf und die Entfernung von Komponentenbeschreibungen angeboten. Die Implementierung beschränkt sich dabei bislang auf die Unterstützung von SMCDL, für deren Verarbeitung die oben angesprochene Java-API zum Einsatz kommt. Über diese können Beschreibungen als Objektmodell repräsentiert werden, welches auf die ontologiebasierte Repräsentation (MCDO) abgebildet wird. Letztere stellt – zusammen mit den darin referenzierten Ontologien, z. B. zur Metadatenrepräsentation – die eigentliche Modellstruktur im CoRe dar, da sie alle vorgestellten Komponentenmodelle subsumiert.
Die folgende Auflistung gibt einen Überblick über die wichtigsten öffentlichen Methoden des CoRe:

SMCDL getMcdlByID (String componentId)
: ... ermöglicht den Abruf einer registrierten SMCDL-Beschreibung über ihre ID.

Components getComponentsBySemantic (String componentSemantic , int amount)
: ... dient dem Abruf von Komponentenbeschreibungen, die eine bestimmte, semantisch beschriebene Funktionalität unterstützen.

Components getComponentsByOperations/Events/Keywords (String() op/ev/kw)
: ... erlaubt die Suche nach Komponenten über vorgegebene Operationen, Ereignisse oder Schlüsselwörter. Diese Möglichkeit wird vorrangig durch Autorenwerkzeuge genutzt.

Boolean register/updateComponent (SMCDL smcd)
: ... realisiert die Registrierung und Aktualisierung von Beschreibungen.

Boolean deleteComponent (String ID)
: ... löscht die Beschreibung mit der angegebenen ID aus dem Modell.

QueryResult doSparqlRequest (String sparql)
: ... erlaubt freie Anfragen an das Modell in SPARQL und liefert die Resultate im SPARQL Query Results XML Format oder in RDF/XML.

Somit kann das CoRe den Aufgaben der Registrierung und Verwaltung nachkommen und die einfache Anbindung durch Autorenwerkzeuge und MREs wird möglich. Der Datenaustausch im Prototyp basiert hauptsächlich auf XML, allerdings kann für die clientseitige Integration die Rückgabe der Komponentenlisten und -beschreibungen je nach Konfiguration auch als JSON-String erfolgen.
Aus Gründen der Komplexität wurde zunächst auf die Umsetzung einer Versionsverwaltung verzichtet, sodass zu einem Zeitpunkt nur eine Version einer Beschreibung registriert sein kann. Eine Erweiterung diesbezüglich ist möglich, impliziert aber die Anpassung der entsprechenden Anfragen von der MRE um die geforderte Version.

Verwaltung von Typdefinitionen

Die Registrierung und Abfrage von Typdefinitionen sowie der entsprechenden Liftings und Lowerings, die im Rahmen der Mediation benötigt werden, erfolgt über einige zusätzliche Web-Service-Methoden. Diese setzen das in Abschnitt 6.2.1 geschilderte Konzept des TyRe ebenfalls auf Basis von Apache Axis2 um.
Die wichtigsten Operationen dieser öffentlichen Schnittstelle umfassen:

String() listKnownURI ()
: ... gibt die URIs aller semantischen Konzepte zurück, für die im TyRe Typdefinitionen, Liftings und Lowerings hinterlegt sind.

Lowering/Lifting getLowering/Lifting (String URI)
: ... gibt Lowering oder Lifting für ein Konzept mit der angegebenen URI zurück.

TypeDefinition getTypeDefinition (String URI)
: ... dient dem Abruf der Typdefinition bzw. des Standard-Grounding-Schemas eines semantischen Konzeptes mit der angefragten URI.

7.2.1.2 Umsetzung der Discovery-Mechanismen

Betrachtet man den in Abschnitt 6.2.3.2 beschriebenen Integrationsprozess und seine Module, so ergeben sich hinsichtlich ihrer Verteilung verschiedene Alternativen. Es stellt sich die Frage, wo die Algorithmen in der vorgestellten Infrastruktur verortet sind, und wie sich dies auf die Robustheit und Performanz der Anwendung auswirkt.

Verteilungsalternativen

Modellinterpretation und Integration obliegen eindeutig dem MRE und müssen plattformspezifisch erfolgen. Somit bieten nur das funktionale, kontextunabhängige

Matching und das nicht-funktionale, aber kontextabhängige Ranking Spielraum zur Verteilung. Ersterem liegt mit der Menge aller registrierten Komponenten das gleiche semantische Modell wie dem CoRe zugrunde. Folglich ist die Verortung der Matching-Algorithmen im CoRe empfehlenswert.
Gleiches gilt prinzipiell für die Ranking-Algorithmen – es sprechen jedoch Gründe für die Entkopplung von Matching und Ranking. Wird letzteres als eigenständiger Dienst umgesetzt, so können vor der Sortierung potentiell mehrere Komponenten-Repositories angebunden werden. Zudem kann die nutzer- bzw. anwendungsspezifische, nicht-funktionale Auswahl auch physisch von der funktionalen Suche getrennt werden. Letztere bleibt dann von jeglicher Kontextanbindung befreit. Dem gegenüber steht allerdings ein stark erhöhtes Datenaufkommen, was durch die nötige Synchronisation der semantischen Modelle bedingt ist, die redundant in den Diensten gehalten werden müssen.
Zur Umsetzung als eigener Dienst stellt die Verortung der Ranking-Algorithmen in den MREs eine Alternative dar, denn dort ist der Kontextbezug direkt gegeben. Doch auch dies führt zu einer hohen Datenlast, da die semantischen Modelle der Kandidaten zum MRE übertragen, dort instanziiert und ausgewertet werden müssten, um das Ranking zu ermöglichen. Gerade für leistungsschwache Clients, z. B. browserbasierte Plattformen auf mobilen Endgeräten, ist dies nicht praktikabel.
Die sinnvollste Alternative – und Wahl für die prototypische Umsetzung – stellt deshalb die Verortung von Matching und Ranking als Teil der Komponentenverwaltung, also im CoRe, dar. Sie erlaubt die Arbeit auf einem gemeinsamen semantischen Modell, weshalb sich der Datenaustausch mit dem MRE auf die einmalige Übertragung der Zielvorlage beschränkt. Auch der Zugriff auf Kontextdaten kann einheitlich erfolgen. Nachteilig erscheint zunächst die Beschränkung auf ein zentrales Repository, allerdings ist die Aggregation der Ergebnisse mehrerer, verteilter Repositories auf Seiten der MREs möglich. Aufgrund der Normalisierung der abschließenden Bewertungen ist die Vergleichbarkeit von Kandidaten verschiedener Repositories gegeben. Die Aggregation macht allerdings einen Redundanzcheck der Kandidatenmengen nötig, um mehrfach registrierte Komponenten herauszufiltern.

Schnittstellen

Mit der Integration von Matching und Ranking in das CoRe, wurde dessen öffentliche Schnittstelle um zwei Methoden erweitert:

Candidates getComponentsByTemplate (IntegrationRequest ir)

> ... nimmt einen IntegrationRequest entgegen, der alle Anfrageparameter zur Suche ausgehend von einer Vorlage enthält (vgl. Abschnitt 6.1.1). Neben dem serialisierten Template zählen dazu Identifikatoren für Plattform und Nutzer, Ranking-Regeln sowie optionale Parameter, wie die Anzahl gewünschter Kandidaten und eine *Blacklist*.
> Das Format der zurückgelieferten Kandidatenmenge, z. B. JSON bei Anfrage der clientseitigen MRE, kann im Request spezifiziert werden

getMatchingResult (String templateID, String componentID, String format)

> ... liefert das MatchingResult für den Kandidaten einer vorangegangen Suchanfrage zurück. Die Zuordnung erfolgt durch die Angabe der Komponenten-ID sowie der ID der Vorlage, die im Rahmen des Discovery-Prozesses vergeben wurde. Auch hier ist die Rückgabe in verschiedenen Formaten möglich.

Wie die bereits vorgestellte Schnittstelle des CoRe basiert die Umsetzung dieser SOAP-Operationen auf dem Axis2-Framework [@Apache, 2011]. Letzteres bietet auch die nötigen Mechanismen zur Sitzungsverwaltung, um Parameter der Anfrage und Ergebnisse im Sitzungskontext ablegen zu können. Dies ist insbesondere für die MatchingResults relevant, die nach der initialen Template-Anfrage gehalten werden müssen, um beim späteren Abruf für den Kandidaten der Wahl direkt bereitzustehen. Bei der Implementierung der Dienstschnittstelle wurde darauf geachtet, dass spätere Erweiterungen, z. B. um eine RESTful-Service-Schnittstelle, ohne Weiteres möglich sind. Eigenheiten der Transportprotokolle wurden deshalb entsprechend gekapselt. Innerhalb des CoRe wurden die beschriebenen Algorithmen wie von Radeck (2011) ausführlich beschrieben in Java umgesetzt und als separate Module im Paket ⁎.integration angelegt. Das Sequenzdiagramm in Abbildung 7.3 verdeutlicht ihren Aufruf bei einer Anfrage über getComponentsByTemplate: Zunächst wird vom CoreService eine neue Instanz der Klasse IntegrationTask erstellt und mit den Daten aus dem Request versorgt. Diese steuert die Discovery und greift dabei auf die Module zum Matching und Ranking zurück. Deren Erstellung erfolgt über eine abstrakte Fabrik nach dem gleichnamigen Muster [Gamma et al., 1995], was den flexiblen Einsatz verschiedener Strategien zur Suche und Sortierung bzw. zur Berechnung des Deckungsgrades ermöglicht.

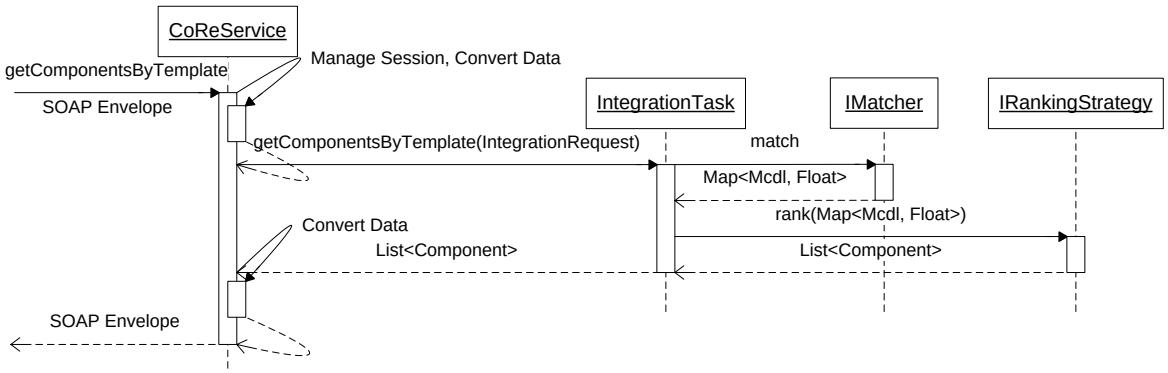

Abb. 7.3: Ablauf der Suche anhand einer Vorlage im CoRe [Radeck, 2011]

Bei der folgenden Anfrage nach dem MatchingResult wird dieses aus dem Sitzungskontext im IMatchingResultStore geladen, zurückgegeben und daraufhin aus dem Speicher entfernt.

Referenzfälle

Im Rahmen der Umsetzung wurden eine Reihe von Test- und Referenzfällen geschaffen, um die Validität und Leistungsfähigkeit der realisierten Algorithmen zu testen. Sie dienen in erster Linie der Analyse des Einflusses der dynamischen Bindung sowie der Extrapolation des zeitlichen Verhaltens. Eine Aussage über typische, praxisnahe Fälle kann indes nicht allgemeingültig erfolgen. Während für öffentliche Repositories – vergleichbar mit *App Stores* – von mehreren Tausend Komponenten hoher Redundanz ausgegangen werden kann, spiegeln die folgenden Fälle die Nutzung innerhalb von Unternehmen wieder. Hier kann von einer überschauberen Anzahl vorrangig interner sowie geprüfter externer APIs ausgegangen werden, die durch Fachexperten komponiert werden können. Zur Einschätzung der Performanz wird in den nächsten Abschnitten auf die folgenden Referenzfälle Bezug genommen:

K100 Es erfolgt eine Anfrage über getComponentByTemplate an ein CoRe mit **100** registrierten Komponenten. Das Template verfügt über acht Properties und eine Operation. Aus der Matching-Phase gehen **sechs** kompatible Komponenten hervor. Auf Basis zweier Occurance-Regeln wird die Kandidatenmenge hinsichtlich des Preises und der Muttersprache des Nutzers gefiltert, was zu **zwei** kompatiblen Komponente führt. Diese werden schließlich durch eine Mapping-Regel im Bezug auf ihren Energieverbrauch gewichtet.

K500 Die obige Anfrage erfolgt an ein CoRe mit **500** registrierten Komponenten, von denen **30** aus der Matching- und **10** aus der Ranking-Phase hervorgehen.

K1000 Die gleiche Anfrage wird an ein CoRe mit **1000** registrierten Komponenten gestellt, von denen **60** aus der Matching- und **20** aus der Ranking-Phase hervorgehen.

Jeder Referenzfall wurde jeweils mit einem, fünf und zehn parallelen Threads durchlaufen, wobei wiederum jeweils 60 Testläufe zur Ermittlung eines Durchschnittswertes durchgeführt wurden. Als Hardware für das Testsystem diente ein 6-Kern Intel Xeon Prozessor mit 2,8 GHz und 8 GB Arbeitsspeicher. Darauf kamen eine 64 bit-Installation von Windows 7 sowie Java 1.6.0_30, der Apache Tomcat 7.0.26 Server und Jena 2.6.3 zum Einsatz.
Die folgenden Abschnitte behandeln realisierten Algorithmen zum Matching und Ranking und greifen zur Bewertung der Leistungsfähigkeit die Referenzfälle auf.

Matching

Die Klasse DefaultMatcher setzt das Matching wie in Abschnitt 6.1.2 beschrieben um. Durch die Erstellung über eine abstrakte Fabrik und den Zugriff durch die Task über die Schnittstelle IMatcher können jedoch alternative Implementierungen einfach hinzugefügt und genutzt werden. So können beispielsweise unterstützte Übereinstimmungsgrade, deren Gewichtung oder die Berechnungsformel für den Deckungsgrad von Komponenten nach Bedarf beeinflusst werden.
In der aktuellen Umsetzung wird die Vorselektion von Kandidaten über eine entsprechende SPARQL-Anfrage an das Modell des CoRe realisiert. Danach erfolgt die Abarbeitung der in Abschnitt 6.1.2 beschriebenen Schritte, also das Matching von Komponenten, Operationen und Ereignissen, einschließlich der Beachtung der *Contravariance* und von Abhängigkeiten. Die korrekte Funktionalität wurde über JUnit [@JUnit.org, 2012] und eine Vielzahl von Testfällen sichergestellt. Die erweiterten Modellkonstrukte der MCDO, insbesondere die Vor- und Nachbedingungen, wurden in der prototypischen Umsetzung zunächst nicht einbezogen, zumal bislang nur die Registrierung von SMCDL-Deskriptoren am CoRe unterstützt wird.
Die Berechnung der bestmöglichen Zuordnung (*bestAssign*) nach dem Matching erfolgt nach der „Ungarischen Methode“ [Kuhn, 1955]. Sie dient der Ermittlung der eindeutigen Zuordnung zwischen zwei Gruppen von Objekten – in diesem Fall zwischen den Schnittstellenbestandteilen von Vorlage und Kandidat – unter Minimierung der „Kosten“. Für jeden Teilschritt können diese Kosten aus den Übereinstimmungsgraden abgeleitet werden, d. h. sie werden negiert: Je höher die Übereinstimmung zwischen zwei Konzepten, desto niedriger fallen Kosten in Form der Mediation aus.

Zur Berechnung kommt die Klasse BestAssignmentSolver zum Einsatz, die auf der praktischen Realisierung von SAWSDL-MX (vgl. Abschnitt 3.1.2) aufbaut. Als Eingabe erhält sie eine Matrix der negierten Übereinstimmungsgrade zwischen den Artefakten der Vorlage und des Kandidaten (In):

$$In = \begin{bmatrix} -3 & -1 & -1 & 0 \\ -1 & 0 & -3 & 0 \\ -1 & -3 & -1 & -1 \end{bmatrix} \qquad Out = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

Im Beispiel sind in der Vorlage drei Artefakte vorgesehen (Zeilen), der Kandidat bietet allerdings vier (Spalten). Es ist leicht zu erkennen, dass das erste Artefakt der Vorlage auf das erste des Kandidaten abgebildet werden sollte, da die Kosten dadurch minimal sind (-3). Zweites und drittes Artefakt sind in Vorlage und Kandidat vertauscht ($Nr.\,2 \mapsto Nr.\,3$ und $Nr.\,3 \mapsto Nr.\,2$). Der BestAssignmentSolver liefert somit die obige Ausgabe Out, in der die Zuordnungen mit dem Wert „1“ markiert sind.

Die Berechnungszeit wird von verschiedenen Faktoren beeinflusst, u. a. von der Anzahl registrierter Komponenten sowie von Umfang und Komplexität der übermittelten Vorlage. Für die oben beschriebenen Referenzfälle betrug die Dauer des Matchings bei einem Thread durchschnittlich 139 ms (K100), 742 ms (K500), und 1480 ms (K1000) und ist erwartungsgemäß direkt von der Größe des Komponentenpools abhängig. Der zeitliche Mehraufwand betrug bei die Nutzung paralleler Threads zwischen 7% und 40% (K100 vs. K1000). Einen Überblick über die Messwerte gibt Abbildung 7.4. Es ist erkennbar, dass die zum Matching benötigte Zeit direkt proportional zur Komponentenmenge steigt. Um die Ladezeiten von Anwendungen in der industriellen Praxis auf akzeptablem Niveau zu halten – nach einer Studie von King (2003) sollte sie maximal 8,6 s betragen – bedarf es deshalb ggf. Optimierungen, z. B. in Form von Caching- oder Verteilungsmechanismen. Derartige Erweiterungen gehen jedoch über die Grenzen dieser Arbeit hinaus.

Ranking

Wie beim Matching wurde auch bei der Umsetzung der in Abschnitt 6.1.3 beschriebenen Sortierungsverfahren auf die Abstraktion und Erweiterbarkeit Wert gelegt. Dazu kommt das Muster *Strategy* [Gamma et al., 1995] bzw. die Klasse AbstractRankingStrategy zum Einsatz, die grundlegende Funktionalität, z. B. für die Auswertung der Bedingungen der Regeln, bietet. Konkrete Strategien zur Sortierung erben von dieser Klasse, wobei das Hauptaugenmerk auf der Umsetzung der MultiplicativeRankingStrategy lag, durch die die Einzelergebnisse der Ranking-Regeln multiplikativ verknüpft werden.

Ausgangspunkt für die Ausführung einer Strategie bilden die anzuwendenden Ranking-Regeln aus dem Kompositionsmodell (vgl. Abschnitt 6.1.3.1), die als Teil der Anfrage zum CoRe – serialisiert in XML – übertragen werden.

Im Rahmen einer studentischen Arbeit wurden verschiedene Alternativen der Serialisierungsformen in SPARQL aus Abschnitt 6.1.3.1 ausführlich evaluiert [Radeck, 2011]. So können in *Occurance Rules* auch `ASK`-Anfragen genutzt werden, bei denen das Metadatum als Platzhalter formuliert und zur Laufzeit durch die URI des Individuums (von `Metadata`) der verarbeiteten Kandidaten ersetzt wird. *Mapping Rules* können ebenso durch `CONSTRUCT`-Anweisungen samt Fallunterscheidung über

den `FILTER`-Ausdruck formuliert werden. Eine Illustration der entsprechenden Serialisierungsformen kann der Arbeit von Radeck (2011) entnommen werden
Die in Abschnitt 6.1.3.1 vorgestellten Serialisierungen wurden für die prototypische Umsetzung gewählt, da sie für alle Testfälle mit Abstand die beste durchschnittliche Berechnungszeit[2] (ohne Auswertung von Vorbedingungen) boten. Bei zehn Kandidaten betrug die zeitliche Differenz pro ausgewerteter Regel lediglich 1–2ms, mit steigender Anzahl an Kandidaten steigert sich der Vorteil pro Regel indes immens. Die durchschnittliche Berechnungszeit zur Discovery einer Occurance-Regel betrug dabei für 87 Kandidaten 18,21 ms gegenüber 42,1 ms in der alternativen Serialisierung. Für die Auswertung einer Mapping-Regel wurden durchschnittlich 3,18 ms benötigt, während die Nutzung von `CONSTRUCT`-Statements ganze 23,07 ms veranschlagt.

Die Einbeziehung der Kontextdaten in die Auswahl- und Filterkriterien erfolgt über *Extension Functions*. Hierzu wurde die Klasse ContextAccessExtensionFunction umgesetzt und über die FunctionRegistry des SPARQL-Subsystems von Jena registriert. Sie wertet Adressierungen der Form `mcdl:queryContext('%path%')` innerhalb der SPARQL-Ausdrücke aus (vgl. Codebeispiel 6.1) und ruft die adressierten Kontextinformationen von CroCo ab.

Im Vergleich zum Aufwand beim Matching fällt die Anwendung der Sortierungsregeln nur marginal ins Gewicht. Abbildung 7.4 zeigt in der Mitte die Messwerte der einzelnen Referenzfälle. Die Filterung und Gewichtung der Kandidatenmenge betrug je nach Anwendungsfall zwischen 2,7 ms für 6 Kandidaten (K100) und 4,9 ms für 60 Kandidaten (K1000). Diese effiziente Verarbeitung liegt u. a. im Einsatz von SPARQL in Verbindung mit dem ausgereiften Jena-Framework begründet. Die tendenzielle Beschleunigung der Verarbeitung bei der Nutzung mehrerer Threads fällt allerdings in den Rahmen von Messschwankungen.

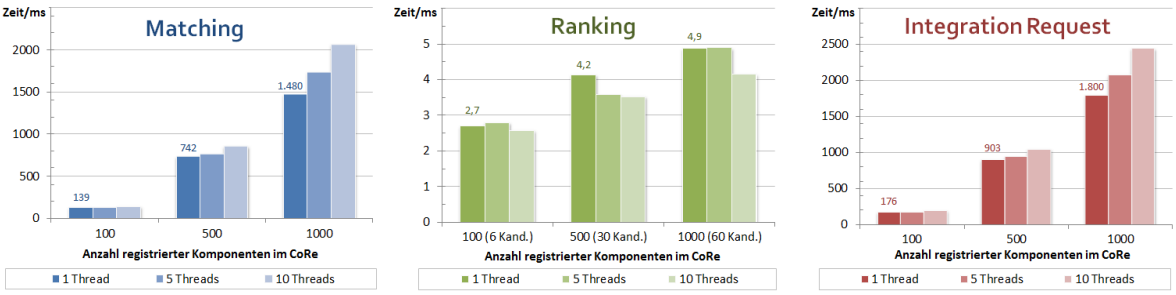

Abb. 7.4: Berechnungs- und Antwortzeiten der Discovery-Algorithmen

Auf der rechten Seite in Abbildung 7.4 ist die vollständige Antwortzeit des CoRe zu erkennen, die mit soapUI 4.0.1 [@*soapUI* 2011] gemessen wurde. Der zeitliche Anteil der Matching-Phase lag dabei zwischen 79 und 84%. Wie bereits angesprochen wurde, besteht hier Handlungsbedarf, da eine höhere Antwortzeit letztlich die Initialisierung der kompositen Anwendung verzögert. Dies kann insbesondere bei komplexen Komponenten zu Problemen führen, da bis zur vollständigen Funktionalität zusätzlich ihre eigene Lade- und Zeichenzeit berücksichtigt werden muss. Positiv ist allerdings die Multithreading-Fähigkeit der Implementierung hervorzuheben. Durch die Möglichkeit der parallelen Anfragen an das CoRe können so mehrere Komponenten

[2]Die Messungen erfolgten auf einem separaten Testsystem, sodass die Vergleichbarkeit mit den Messwerten der Discovery nicht gegeben ist.

gleichzeitig gesucht und im Anschluss integriert werden. Die Integration von zehn Komponenten für den Fall K500 kann durch die Parallelisierung beispielsweise in 12% der sequentiellen Integrationszeit erfolgen.[3]

Zusammenfassend lässt sich feststellen, dass die in Abschnitt 6.2.3.2 vorgestellten Konzepte zur dynamischen Suche und Auswahl von Mashup-Komponenten vollständig umgesetzt werden konnten. Grundlage dafür bilden die Dienste zur semantischen Verwaltung verfügbarer Komponenten (CoRe) und Datentypen (TyRe). Ganz im Sinne einer SOA können diese plattformübergreifend, d. h. von verschiedenen MREs genutzt werden. Den verschiedenen Ausprägungen dieser Laufzeitumgebungen widmet sich der folgende Abschnitt.

7.2.2 Kompositions- bzw. Laufzeitumgebungen

Die in Abschnitt 6.2.2 vorgestellte Referenzarchitektur wurde in verschiedenen Implementierungen umgesetzt, um u. a. die Plattformunabhängigkeit der modellierten Anwendungen validieren zu können.
Schwerpunkt bei der Prototyp-Entwicklung bildete ein MRE, welches gemäß der Prinzipien der *Thin Server Architecture* [@Prasad et al., 2007] die Modellinterpretation und Ausführung der Anwendung komplett clientseitig im Browser realisiert. Neben der Auslagerung einiger Module auf den Server wurde zudem eine eigenständige, serverseitige Laufzeitumgebung auf Basis der Eclipse Rich Ajax Platform (RAP) [@Eclipse, 2011b] entwickelt. Zur Anwendung der Konzepte im Kontext von Geschäftsprozessen erfolgt schließlich die Erweiterung einer WS-HumanTask-Engine, um die Verarbeitung menschlicher Aktionen in BPEL4People-Prozessen mit Hilfe kontextadaptiver, kompositer Benutzerschnittstellen zu ermöglichen.
Die folgenden Abschnitte stellen die drei genannten MREs kurz vor, wobei die clientseitige Thin Server Runtime (TSR) den Schwerpunkt der Betrachtung bildet. Daneben wird ein Überblick über die serverseitige Lösung auf Basis von Eclipse RAP und die Integration in eine Human-Task-Engine gegeben.

7.2.2.1 Clientseitige Komposition mit der Thin Server Runtime (TSR)

Das funktional umfangreichste MRE wurde u. a. im Rahmen diverser studentischen Arbeiten nach den Regeln der *Thin Server Architecture* [Prasad et al., 2007] umgesetzt und kontinuierlich weiterentwickelt (vgl. Waltsgott (2009)). Es ermöglicht die vollständig clientseitige Komposition und Ausführung und bindet alle dazu nötigen Infrastrukturdienste, z. B. zur Komponenten- und Kontextverwaltung, direkt an.
Grundlage für die Umsetzung bildete das JavaScript-Framework Ext Core 3.3 [@Sencha, 2012], welches die Entwicklung clientseitiger Webanwendungen durch diverse Abstraktionen und ein eigenes Klassenmodell unterstützt. Mit der MIT-Lizenz ist es zudem weniger reglementiert als das umfangreichere UI-Framework aus gleichem Hause. Alle in Abschnitt 6.2.2 beschriebenen Module wurden als individuelle Klassen implementiert. Im Folgenden werden einige Aspekte der Ausführung exemplarisch herausgegriffen und ihre Umsetzung genauer beleuchtet.

[3] Sequentieller Abruf: $10\ Komponenten * 903\,ms = \mathbf{9030\,ms}$; Paralleler Abruf mit zehn Threads: $10\ Komponenten * 1047\,ms/10\ Threads = \mathbf{1047\,ms}$.

Initialisierung

Zum Start einer Mashup-Anwendung ruft der Client wie gewöhnlich deren Webseite auf. Dieses HTML-Dokument enthält den Initialisierungscode der TSR sowie Verweise auf das zu interpretierende Kompositionsmodell. Ebenfalls enthalten sind optionale Informationen zur Anwendungskonfiguration, wie die Adresse des CoRe, des Kontextdienstes, usw.
Zunächst wird der Application Manager geladen, welcher in Folge alle weiteren Module der TSR initialisiert. Daraufhin wertet er das angeforderte Kompositionsmodell aus und fordert die referenzierten Mashup-Komponenten bzw. Kandidaten vom CoRe über dessen SOAP-Schnittstelle ab. In der vorliegenden Umsetzung der TSR wird bei Template-Anfragen stets die Komponente mit der höchsten Wichtung aus der Discovery gewählt und integriert. Integration, Zugriff und lokale Verwaltung von Komponenten erfolgen schließlich unter Verwendung der im SMCDL-Binding angegebenen Informationen, wie in Abschnitt 6.2 beschrieben.
Codebeispiel 7.4 zeigt eine beispielhafte Umsetzung der Life-Cycle-Methode init() zur Initialisierung einer Mashup-Komponente entsprechend der Darstellung in Abschnitt 6.2.3.1. Sie wird indirekt vom Component Manager über den Wrapper aufgerufen und erhält als Aufrufparameter ein ComponentContext-Objekt (Zeile 2), welches der Komponente Referenzen auf diverse Module des MRE bietet (Zeile 4–6). Diese werden i. d. R. für den späteren Zugriff an lokale Variablen gebunden. Die Implementierung der Methode obliegt Komponentenentwicklern. Im Sinne des Life-Cycle-Managements muss nach Abschluss der Initialisierung das MRE benachrichtigt werden (Zeile 9).

```
1  // Aufrufparameter vom Typ 'ComponentContext'
2  init: function(ctx){
3      // Auslesen der Referenzobjekte des MRE
4      this.log = ctx.getAttribute("Logger");
5      this.broker = ctx.getAttribute("EventHandler");
6      this.style = ctx.getAttribute("Style");
7      ...
8      // Anzeigen der erfolgreichen Initialisierung durch Life-Cycle-Event
9      broker.publish(initMessage);
10 }
```

Lst. 7.4: Beispiel der Life-Cycle-Methode zur Initialisierung einer Komponente

Kommunikation

Zur Umsetzung des in Abschnitt 6.2.4 vorgestellten Kommunikationssystems kommt in der TSR der Singleton EventBroker zum Einsatz, der eine Erweiterung der Klasse Ext.util.Observable aus dem Ext Framework darstellt. Letztere bietet grundlegende Funktionalitäten zur Publikation und Verarbeitung von Ereignissen. Intern verwaltet der Broker Kommunikationskanäle (Klasse Channel) in einem assoziativen Array. Jeder Kanal repräsentiert einen Link, BackLink oder PropertyLink aus dem Kompositionsmodell und ist durch die Kombination aus Name und Signatur eindeutig bestimmt. Weiterhin existieren dedizierte Kanäle für System-, Life-Cycle- und Fehlerereignisse.

Die wichtigsten Methoden des Event Brokers umfassen:

addChannel (name , type , dataTypes , threshold)
: . . . wird zum Anlegen eines neuen Kanals mit dem Namen *name* aufgerufen. Der

Parameter $type$ gibt an, ob es sich um einen Link, BackLink oder PropertyLink handelt. $dataTypes$ enthält die Signatur des Kanals als Liste von Typen bzw. URIs. Der optionale Parameter $threshold$ gibt schließlich die zeitliche Begrenzung zur Aktualisierung beim BackLink an.

addEventToChannel (cId , name , dataType , channelName , callbackId , scope)
: ...registriert ein neues Ereignis mit dem Namen $name$ und der Signatur $dataType$ der Komponente mit der ID cId als Publisher am Kanal $channelName$. Falls es sich um einen BackLink handelt, wird die $callbackId$ ihrer Rückrufoperation übergeben. Handelt es sich beim Sender um keine Komponente, sondern z. B. um ein Modul des MRE, bleibt cid unbelegt.

subscribe (cId , name , dataType , channelName , handler, scope , callbackId)
: ...registriert eine Operation mit dem Namen $name$ und der Signatur $dataType$ der Komponente mit der ID cId als Subscriber am Kanal $channelName$. Falls der Empfänger keine Komponente ist, kann die aufzurufende Methode als $handler$ übergeben werden. Handelt es sich um einen BackLink, wird außerdem das zur Operation zugehörige Rückgabeereignis über die $callbackId$ angegeben.

publish (cId, message)
: ...dient der Publikation von Ereignissen unter Angabe der auslösenden Komponente (cId) und der eigentlichen Nachricht $message$.

Im Vergleich zum Aufruf in Codebeispiel 7.4, Zeile 9, enthält die *publish*-Methode hier einen zusätzlichen Parameter – die plattforminterne ID der Komponente. Diese ist nötig, um die Nachricht eindeutig zuordnen zu können, da sich Ereignisnamen in verschiedenen Komponenten gleichen können. Da die Komponente ihre eindeutige ID im Anwendungskontext nicht kennt, kommt wiederum der Wrapper ins Spiel. Als Mittelsmann der Kommunikation zwischen Komponente und Event Broker fügt er diese Information hinzu. Dazu stellt jeder Wrapper selbst eine *publish()*-Methode bereit und agiert aus Sicht der Komponenten als Event Broker.

```
1  ...
2  // Erstellung der Nutzdaten
3  var value = '<stock>...</stock>';
4  // Erstellung einer neuen Nachricht \ddot{u}ber das MRE
5  var message = broker.createMessage();
6  // Konfiguration der Nachricht
7  message.setName('currentStockValue');
8  message.setCallbackId(request.getCallbackId());
9  message.appendToBody('stockresponse',value);
10 // Absenden der Nachricht
11 this.broker.publish(message);
```

Lst. 7.5: Nachrichtenerstellung und -versand durch eine Komponente

Das in Abschnitt 6.2.4 vorgestellte Nachrichtenobjekt können Komponenten über den Event Broker erstellen und darauf, wie Codebeispiel 7.5 zeigt, über Getter- und Setter-Methoden zugreifen. Im Beispiel wird eine neue Nachricht mit dem Namen „currentStockValue“ erstellt, der den Ereignisnamen aus dem Modell widerspiegelt (Zeile 5–7). Da die Nutzdaten als Reaktion auf eine Anfrage gesendet werden, wird die *CallbackId* aus der Anfrage in die Nachricht übernommen (Zeile 8). Für die Methode setBody() existieren zwei Implementierungen. Im Beispiel wird der Nachricht ein einzelner Parameter angehangen (Zeile 9), es kann jedoch auch ein assoziatives

Array übergeben werden, welches alle spezifizierten Parameter enthält. Über den Aufruf von publish in Zeile 11 wird die Nachricht schließlich gesendet.

Mediation

Wie im Rahmen der Konzeption beschrieben, ist bei der Kommunikation zwischen semantisch kompatiblen Komponenten u. U. die Vermittlung auf syntaktischer bzw. Datenebene nötig. Diese Mediation muss durch das MRE bei der Nachrichtenvermittlung erfolgen, wobei auf die Typinformationen, Liftings und Lowerings der vermittelten Konzepte über das TyRe (vgl. Abschnitt 6.2.1) zugegriffen wird.

Die Mediation, d. h. die Auflösung semantischer Subklassenbeziehungen, macht die Verwaltung eines potentiell großen, ontologiebasierten Modells nötig. Den dadurch implizierten Leistungsanforderungen sind browserbasierte MREs, wie die TSR, im Normalfall nicht gewachsen – zumindest ist mit erheblichen Leistungseinbußen zu rechnen. Deshalb wurde das Mediation-Modul in einen Web Service ausgelagert. Dieser kann – ebenso wie CoRe und TyRe – von allen Laufzeitumgebungen genutzt werden und setzt semantische Casts von Daten unter der Angabe von Ausgangs- und Zielkonzept um. Wie die bereits vorgestellten Dienste erfolgte die Realisierung als SOAP-Service auf Basis von Apache Axis2 unter Anbindung des TyRe. Die wichtigste, öffentliche Operation namens convert nimmt den Cast vor und erwartet als Aufrufparameter *data* – die Nutzdaten – sowie URIs der Ausgangs- und Zielklassen. Als Rückgabe liefert der Mediation Service die Nutzdaten in der syntaktischen Repräsentation der Zielklasse.

Drag-and-Drop

Für die Realisierung der Drag-and-Drop-Interaktion wird jede UI-Komponente in ein Ext.Panel des Ext-Frameworks gezeichnet. Diesem kann eine *DropZone* hinzugefügt werden, die bei Bedarf sämtliche Drop-Events auf der Komponente abfängt. Wird nun, wie in Abschnitt 6.2.4 beschrieben, ein *Drag*-Event von einer Komponente signalisiert, so werden die DropZones aller „kompatibler" Komponenten aktiviert.

Auch die DropZones sind als *Panel* realisiert, die auf DOM-Level-2-Ereignisse wie *mouseup*, *mouseover* und *mouseout* reagieren.[4] Im Hinblick auf die verstärkt eingesetzte Drag-and-Drop-API aus HTML5 [@Hickson, 2012], werden gleichermaßen deren Ereignisse überwacht (*dragcenter*, *dragover*, *dragleave*, *drop*).

Wenn Daten über eine Komponente „fallen gelassen" werden, wird die Suche nach passenden Operationen in der darunter liegenden Komponente angestoßen. Die Nutzdaten werden daraufhin durch den Aufruf von getDragData von der Ausgangskomponente abgefragt und an die passende Operation gesendet. Sind mehrere Empfänger möglich, wird dem Nutzer ein Popup-Fenster zur Auswahl angeboten. Danach werden alle DropZones deaktiviert.

Dienstzugriff

Für den Zugriff auf Dienste im Backend kommt – gleichsam für Komponenten wie für die Module der TSR – die Service-Access-Schicht zum Einsatz. Diese leitet alle Anfragen über einen Proxy innerhalb der Anwendungsdomäne, um die Beschränkungen aufgrund der Same-Origin-Policy [@Zalewski, 2011] zu umgehen. Ein entsprechender Proxy wurde als Java Servlet umgesetzt.

[4] Aus *mouseover* und *mouseup* wird somit *drop* geschlussfolgert.

Die Klasse ServiceAccess – für Komponenten über den ComponentContext erreichbar – bietet die Möglichkeit, ein XMLHttpRequest (XHR)-Objekt für den Dienstzugriff anzulegen. Dazu wurde die vom W3C spezifizierte XHR-Schnittstelle [@Kesteren, 2012] vollständig umgesetzt. Aufgrund ihrer Standardisierung kann sie plattformübergreifend auch in anderen Technologien genutzt werden.
Codebeispiel 7.6 zeigt den praktischen Einsatz innerhalb der TSR. Über das ServiceAccess-Objekt wird in Zeile 6 ein neues XHR-Objekt erstellt und als SOAP-basierte POST-Anfrage an eine vorher definierte Zieladresse (*_serviceUrl*) konfiguriert (Zeile 7–8). In Zeile 10–14 wird die Callback-Methode zur Behandlung der zurück gelieferten Antwort spezifiziert, und in Zeile 15 die Anfrage schließlich gesendet.

```
1  // Referenz auf das ServiceAccess-Objekt
2  var _sa= componentContext.getAttribute("ServiceAccess");
3  // Erstellung der Nachricht
4  var msg = '<soap:envelope> ... </soap:envelope>';
5  // Erstellung eines neuen Request-Objektes
6  var xhr = _sa.createXHR();
7  xhr.open('POST', _serviceUrl);
8  xhr.setRequestHeader('Content-Type', 'application/soap+xml');
9  // Callback-Methode
10 xhr.onreadystatechange = function(xhro) {
11     if (xhro.readyState == 4 && xhro.status == 200) {
12         // Behandlung der Antwort
13     }
14 };
15 xhr.send(msg);
```

Lst. 7.6: Dienstzugriff mittels XHR über die Service-Access-Schnittstelle

Weitere ausgewählte Funktionalitäten

Zur Sicherung von Komponenten- und Anwendungszuständen wurde auf clientseitige Persistenzmechanismen zurückgegriffen. Da sich diese von Browser zur Browser unterscheiden, wurde die PersistJS-Bibliothek [@Duncan, 2007] zur Abstraktion eingesetzt. Sie bietet eine einheitliche Schnittstelle zur clientseitigen Zustandsicherung – z. B. in Fehlerfällen oder beim Komponententausch – und nutzt dabei die jeweils verfügbare Browserimplementierung, wie die inzwischen stark verbreitete HTML5-Spezifikation *Web Storage* [@Hickson, 2011].
Die Persistierung selbst umfasst die Extraktion des Anwendungszustandes – repräsentiert durch existierende Komponenten, Kommunikationskanäle, Sichtbarkeiten, etc. – sowie der Komponentenzustände – repräsentiert durch die Gesamtheit aller Komponenteneigenschaften. Danach erfolgt ihre Speicherung mittels PersistJS. Beim nächsten Anwendungsstart kann überprüft werden, ob eine Sicherung aus einer zuletzt aktiven Sitzung besteht, die der Nutzer laden kann.

Zur Steigerung der Lade- und Ausführungsgeschwindigkeit kommt der Yahoo! UI Compressor [@Yahoo!, 2011b] zum Einsatz. Er wird genutzt, um die gesamte Codebasis der TSR vor der Auslieferung an den Client zu minimieren und zu optimieren. Der Umfang von etwa 540 KB für alle Module der TSR (ohne externe Bibliotheken) kann so auf nur 240 KB verringert werden. Gleichzeitig verbessern sich Initialisierungszeit und Ansprechverhalten zur Laufzeit.

Auch für Entwicklung und Tests wurden Funktionalitäten eingebaut. Die Integration von log4javascript [@Down, 2011] erlaubt beispielsweise die clientseitige Protokollierung von Log-Ausgaben in ein separates Browserfenster und gleicht aus Sicht

der Entwickler den aus der traditionellen Software-Programmierung bekannten Verfahren, z. B. hinsichtlich der Unterstützung verschiedener Log-Level (`INFO`, `WARN`, `ERROR`, etc.). Der Logger ist als Modul der TSR vorhanden und steht Modulen sowie Komponenten über den ComponentContext zur Verfügung.

Insgesamt stellt die TSR die umfangreichste Implementierung eines MRE dar. Durch ihre erfolgreiche Umsetzung und die Tests mit den in Abschnitt 7.3 vorgestellten Beispielanwendungen konnte das Konzept der dynamischen, clientseitigen Komposition und Ausführung von Mashup-Anwendungen nachgewiesen werden. Wo es möglich und sinnvoll erschien, wurden Module in dedizierte Backend-Dienste ausgelagert, z. B. in Form der Mediation. Ansonsten stützt sich die TSR auf die im letzten Abschnitt vorgestellten Dienste zur Verwaltung und Discovery.

7.2.2.2 Serverseitige Komposition auf Basis von Eclipse RAP

Zur serverseitigen Komposition wurde eine alternative Laufzeitumgebung auf Grundlage von Eclipse RAP [@Eclipse, 2011b] entwickelt und im Rahmen des umgebenden Forschungsprojektes in die Plattform CAS Open [@Niemann, 2009] eines Industriepartners integriert.

Im Gegensatz zur TSR kamen bei der Umsetzung keine clientseitigen Technologien zum Einsatz. Zwar nutzt RAP zur Darstellung der Anwendungsoberfläche im Browser das JavaScript-Framework Qooxdoo [@Qooxdoo, 2011] – es dient jedoch allein der clientseitigen Repräsentation der eigentlichen, serverseitigen Objekte des RAP Widget Toolkit (RWT). Diese werden, wie die gesamte Plattform auch, mit Java umgesetzt. Der Lebenszyklus eines RWT-Widgets ist von der permanenten Synchronisation zwischen Client und Server bestimmt. Veränderungen auf der Serverseite werden über *Life-Cycle-Adapter* an den Client kommuniziert, und gleichsam werden Interaktionen auf dem Client an den Server übermittelt. Die Anwendungs- bzw. Präsentationslogik befindet sich somit allein auf der Serverseite, d. h. auch die Verwaltung, Kommunikation und Koordination zwischen Widgets finden dort statt.

Für die Anwendung der vorgestellten Konzepte bedeutet die Nutzung von RAP zunächst, dass Komponenten serverseitig und in Java vorliegen müssen. Es besteht jedoch die Möglichkeit, eigene Widgets zu erstellen, die aus den clientseitigen sowie zwei serverseitigen Ressourcen – Life-Cycle-Adapter und Widget – bestehen. Deshalb wurde ein Mechanismus entwickelt, der diese serverseitigen Artefakte dynamisch aus der MCDL-Beschreibung generiert. So können clientseitige Komponenten, wie sie bei der TSR integriert werden, auch mit der RAP-Umgebung zum Einsatz kommen. Sie werden als clientseitige Widgets gekapselt und kommunizieren dann über das generierte Life-Cycle-Adapter und die serverseitige Repräsentation.

Abbildung 7.5 verdeutlicht dies am prinzipiellen Aufbau des RAP-basierten MRE. Im oberen Bereich ist die Aufteilung der kompositen Anwendung einen client- und einen serverseitigen Teil zu erkennen. Wird eine Mashup-Komponente integriert, die nativ für die Plattform entwickelt wurde (ganz links), nutzt sie ein Life-Cycle-Adapter (LCA) und die native clientseitige Repräsentation der Plattform. Zur Integration clientseitiger Komponenten (daneben) werden die gelb eingefärbten Artefakte, d. h.

die serverseitige Repräsentation und der Wrapper auf Clientseite, generiert. Native Service-Komponenten (ganz rechts) können hingegen allein auf dem Server ausgeführt werden.

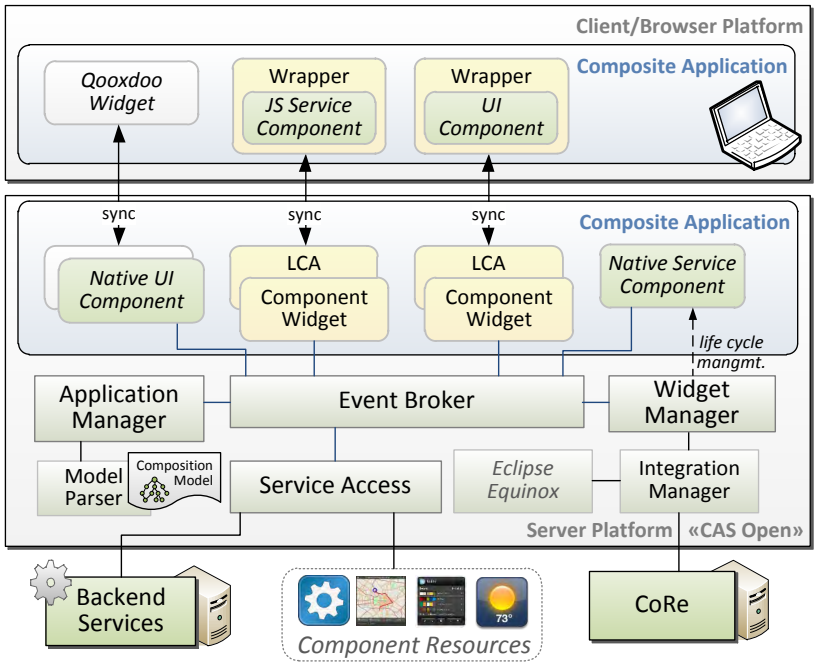

Abb. 7.5: Client-Server-Verteilung bei der RAP-basierten Laufzeitumgebung

Die Ausführungsumgebung für RAP bildet Equinox [@Eclipse, 2012a], eine Referenzimplementierung der Open Services Gateway initiative (OSGi)-R4-Spezifikation und im unteren Teil der Abbildung angedeutet. OSGi definiert ein Komponentenmodell, welches die Modularisierung von Software-Komponenten und deren dynamische (De-)Installation erlaubt. Dies kann für die Repräsentation und dynamische Integration von Mashup-Komponenten ausgenutzt werden, allerdings bedarf es dazu ihrer Kapselung als OSGi-*Bundles*. Native, serverseitige Mashup-Komponenten müssen durch Entwickler deshalb als Java ARrchive (JAR)[5] bereitgestellt werden, während bei der automatischen Kapselung clientseitiger Komponenten der o. g. Mechanismus diese Aufgabe übernimmt.
Bei der Initialisierung einer komposiiten Anwendung erfolgt die Interpretation des Kompositionsmodells durch den Application Manager. Er ruft zunächst über die vorgestellte SOAP-Schnittstelle des CoRe für alle benötigten Komponenten deren Beschreibungen bzw. die Kandidatenlisten ab. Die Kapselung clientseitiger Komponenten als OSGi-Bundles erfolgt im CoRe und für die Laufzeitumgebung transparent. Dazu wurde ein zusätzliches Modul entwickelt und in den bereits vorgestellten *Integration Task* (vgl. Abschnitt 7.2.1) eingebunden. Nähere Informationen hierzu bieten Rümpel et al. (2010).
Somit werden nur Kandidaten an das MRE ausgeliefert, die technologisch kompatibel sind. Nach der Auswahl einer zu integrierenden Komponente aus einer Kandidatenliste wird zunächst geprüft, ob sie bereits im lokalen Cache existiert bzw. als Bundle registriert ist. Damit kann dem mehrfachen Abruf von Komponenten für verschiedene Client Sessions vorgebeugt werden. Im Fall neuer Komponenten werden

[5]Die JAR-Datei wird in der MCDL-Beschreibung wie üblich als *Dependency* angegeben.

die entsprechenden JAR-Bundles abgerufen und mit Hilfe von OSGi „Declarative Services“ [@OSGi, 2007] geladen. Die dazu nötige, XML-basierte Bundle-Beschreibung wird bei der Generierung aus der (S)MCDL erstellt.
Das MRE ist selbst als Bundle am OSGi-Framework registriert und kann somit auf die Installation neuer Bundles reagieren. Über deren öffentlichen Schnittstellen besteht u. a. Zugriff auf alle in der (S)MCDL definierten Artefakte, z. B. verfügbare Events und Operationen, Metadaten und enthaltene Ressourcen. Letztere werden – wie in der TSR – zentral verwaltet, um Redundanzen zu vermeiden. Nutzen beispielsweise mehrere Bundles die Silverlight-JavaScript-Bibliothek, so wird diese nur einmal auf dem Client geladen. Neben erhöhter Performanz können dadurch ungewollte Komplikationen, z. B. Namensraumkollisionen, vermieden werden.
Auch die Initialisierung von Komponenten-Bundles erfolgt über deren öffentliche Schnittstellen. Als Rückgabewert erhält das MRE das RWT-Widget selbst, welches im Fall einer UI-Komponente entsprechend platziert werden kann. Die Anwendungsoberfläche wird dazu im Rahmen der Modellinterpretation mit RAP-Bordmitteln erstellt und mit Platzhaltern versehen. Diese werden schrittweise mit Komponenten gefüllt, sobald deren Installation abgeschlossen ist.
Eine besondere Rolle kommt dem Application Manager zu. Neben der Initialisierung und Integration repräsentiert er die verwaltete Komposition nach außen hin als eigenständiges Bundle und realisiert die öffentliche Schnittstelle zu einer potentiellen Rahmenanwendung. Im konkreten Anwendungsfall – Abbildung 7.13 zeigt auf der rechten Seiten einen Screenshot einer kompositen Customer Relationship Management (CRM)-Anwendung – erfolgte u. a. die Integration in die Lösung CAS PIA [@CAS, 2011]. Diese stellt die Daten für die Komposition und wird im Modell als *Environment* repräsentiert, wie in Abschnitt 5.2.1.3 beschrieben.

Mit der Umsetzung der serverseitigen Laufzeitumgebung wurde gezeigt, dass die Modellierung kompositer Anwendungen prinzipiell unabhängig von Zielplattform und Verortung der Komposition auf Client oder Server erfolgen kann. Vielmehr können mit Hilfe der angesprochenen Erweiterungen des *Integration Services* sogar clientseitige Komponenten serverseitig integriert werden. Neben der Basisfunktionalität, wie sie bereits durch die TSR unterstützt wird, konnte mit der RAP-basierten Lösung die Verknüpfung von Komposition und serverseitiger Rahmenanwendungen veranschaulicht und getestet werden. Eine besondere Rolle kommt diesem Konzept bei der Integration in Geschäftsprozesse zu, wie im nächsten Abschnitt beschrieben.

7.2.2.3 Kompositionsplattform zur Interaktion mit Geschäftsprozessen

Bei der Betrachtung des verwandter Arbeiten wurde deutlich, dass im Bereich der Interaktion mit Diensten und Geschäftsprozessen trotz aller Standardisierungsbemühungen bislang nur unzureichende Erfolge zu verzeichnen sind. Die in Abschnitt 3.1.1 vorgestellten Spezifikationen BPEL4People und WS-HumanTask (WS-HT) ermöglichen zwar die Integration menschlicher Aktivitäten in Geschäftsprozesse, beschränken sich aber auf Definition von Rollen und die Beschreibung der Interaktion mit dem umgebenden Prozess. Die in dieser Arbeit entwickelten Konzepte ermöglichen es nun, die Benutzerschnittstelle einer *Human Task* als universelle Komposition zu modellieren, die im Hintergrund mit dem Geschäftsprozess kommuniziert.

Zur Unterstützung dieser Vision wurde im Rahmen einer Diplomarbeit ein MRE entwickelt und mit einer WS-HT-Engine gekoppelt [Voigt, 2009] .
Zur Integration von Task und UI-Komposition wird das Element `rendering` der WS-HT-Beschreibung genutzt, dessen Inhalt durch die WS-HT-Spezifikation nicht vorgeschrieben wird. Im Normalfall enthält es eine proprietäre Beschreibung, die von Task List Clients (TLCs) ausgelesen wird und zur Darstellung einzelner Tasks führt. In der vorliegenden Implementierung wird das Rendering zur Integration oder Referenzierung eines Kompositionsmodells in seiner Serialisierung als XMI genutzt. Zur Verarbeitung wurde die HumanTask-MRE geschaffen, das sich nahtlos in die Ausführungsumgebung von WS-HT-basierten Prozessen integriert. Abbildung 7.6 veranschaulicht diese Integration von Mashup-Kompositionen zur Interaktion (grün) mit der Geschäftsprozessinfrastruktur (rot).

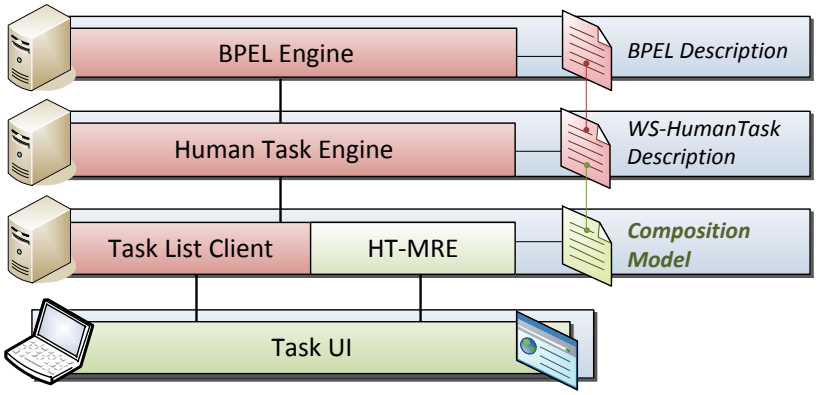

Abb. 7.6: Integration von Kompositionen zur Interaktion in Geschäftsprozessen

Die Umsetzung des HT-MRE erfolgte auf Basis der Open-Source Geschäftsprozess-Engine ActiveBPEL 5 – inzwischen ActiveVOS [@ActiveVOS, 2011] – und erweitert dessen TLC um Mechanismen zur Interpretation und Ausführung von Mashup-Kompositionen. Die Erweiterungen wurden hauptsächlich serverseitig in Java vorgenommen und betreffen die Interpretation der Renderings. Je nach Rendering-Typ erfolgt die Verarbeitung der plattform- bzw. herstellerspezifischen Anweisungen – im Fall von ActiveBPEL XSLT – oder des Mashup-Kompositionsmodells. Der dafür geschaffene Parser und die serverseitigen Klassen realisieren die Generierung von clientseitigem Code zur Komposition und dessen Auslieferung. Im Gegensatz zur Umsetzung für RAP erfolgt die Bindung der Komponenten erst beim Client, wobei auf Teile der TSR zurückgegriffen wurde. Die angebundenen Infrastrukturdienste entsprechen den bereits dort vorgestellten, sodass hier auf eine ausführlichere Darstellung verzichtet wird.

Eine besondere Herausforderung bei der Umsetzung stellte die Kommunikation mit dem Geschäftsprozess dar. In WS-HT sind Tasks durch ein- und ausgehende Nachrichten charakterisiert, die einem vordefinierten Schema entsprechen. Die Kommunikation mit dem Prozess kann somit nicht pro Komponente erfolgen, sondern das „Ergebnis“ der Interaktion muss aggregiert und schemakonform an den Geschäftsprozess zurückgegeben werden. In der HT-MRE wird dazu eine dedizierte UI-Komponente genutzt, die die globale Datensenke für eine Komposition darstellt. Sie muss spezifisch für eine Task konfiguriert werden, was in der vorliegenden Implementierung durch die Parametrisierung mit einem XSLT-Stylesheet erfolgt, welches

die Kombination der individuellen Eingabedaten zur gewünschten Ausgabenachricht umsetzt. Aus Nutzersicht gleicht die Komponente eine Statusanzeige, die Auskunft darüber gibt, ob alle zum Abschluss der Task nötigen Eingaben vorhanden sind. Ist dies der Fall, bietet sie eine Schaltfläche zum Abschließen der Human Task.

Aufgrund der hohen Komplexität kann die Umsetzung hier nur angerissen werden. Die zugrunde liegenden Konzepte und weiterführende Informationen wurden jedoch ausführlich in [Pietschmann et al., 2009a] vorgestellt. Der interessierte Leser findet dort ebenfalls Beispielszenarien aus den zu Testzwecken entwickelten Prozessen und integrierten Kompositionen. Im Rahmen der Tests konnte gezeigt werden, dass das Konzept praktikabel und tragfähig ist. Die entstehenden Oberflächen zur Interaktion und Bewältigung der menschlichen Aufgabe heben sich durch ihre Reichhaltigkeit und Interaktivität vom konventionellen Lösungen ab. Der Aufwand der Erstellung bleibt dabei mindestens gleich und sinkt gegenüber der manuellen, deklarativen Beschreibung von Task-UIs sogar, da auf bestehende Mashup-Komponenten und -Kompositionen zurückgegriffen werden kann. Die Entwicklung von HumanTask-basierten Oberflächen kann durch die Nutzung der Konzepte plattformunabhängig, schnell und kostengünstiger erfolgen. Die Lücke zwischen der Prozess- und der UI-Modellierung wird geschlossen. Der entwickelte Prototyp bildet somit die Grundlage für die modellgetriebene Entwicklung interaktiver Geschäftsprozesse, macht jedoch eine stärkere Verzahnung der individuellen Modelle (BPEL, BPEL4People, WS-HumanTask und Kompositionsmodell) nötig.
Mit der Vorstellung dieser dritten Laufzeitumgebung zur Mashup-Komposition ist die Betrachtung der prototypisch umgesetzten MREs abgeschlossen. Es konnte gezeigt werden, dass Anwendungen auf Basis des Kompositionsmodells prinzipiell unabhängig von der Ausführungsumgebung modelliert werden können, was nicht zuletzt die Wiederverwendbarkeit von Komponenten und Kompositionen in verschiedenen Kontexten erlaubt. Die Infrastrukturdienste zur Verwaltung und dynamischen Suche von Komponenten bleiben indes plattformunabhängig und können von allen MREs genutzt werden. Den letzten Schwerpunkt der Betrachtung bildet die Realisierung der Adaptionskonzepte, auf die im nächsten Abschnitt eingegangen wird.

7.2.3 Kontextverwaltung und Adaptionsmechanismen

Die in Abschnitt 6.3 vorgestellten Konzepte zur Kontextualisierung bzw. dynamischen Adaption kompositer Anwendungen wurden ebenfalls prototypisch umgesetzt und validiert. Grundlage dafür bildete die Realisierung des Web Services CroCo zur Kontextverwaltung. Da die TSR-Umgebung die am weitesten entwickelte und ausgereifteste MRE darstellt, wurde das Adaptionssystem als dessen Erweiterung, d. h. als clientseitige Lösung realisiert. Die folgenden Abschnitte geben einen Überblick über die Ergebnisse der prototypischen Realisierung.

7.2.3.1 Der Kontextdienst CroCo

Basierend auf den in Abschnitt 6.3.1.1 vorgestellten Konzepten wurde der ontologiebasierte Kontextverwaltungsdienst CroCo in Java umgesetzt (vgl. Winkler (2007)). Das semantische Kontextverwaltungssystem greift zur Verarbeitung der RDF- und OWL-Modelle auf das bereits angesprochene Semantic Web Framework

Jena [@Jena, 2011] zurück. Es kommt unter anderem zur Konsistenzprüfung und zur Ableitung höherwertigen Wissens zum Einsatz. Der dafür genutzte Reasoner basiert auf dem `GenericRuleReasoner` von Jena, welcher Inferenz auf Basis von *Jena Rules* ermöglicht. Letztere können in separaten Dateien gehalten, unabhängig gewartet und erweitert werden. Zur Persistierung der konsistenten und geschlussfolgerten Kontextdatenbasis wird eine MySQL Datenbank eingesetzt.
Für die Kopplung mit der restlichen Infrastruktur verfügt das Kontextverwaltungssystem über eine Web-Service-Schnittstelle, die mit Apache Axis2 [@Apache, 2011] umgesetzt wurde. *Update* und *Query Service* ermöglichen die Übermittlung neuer Kontextdaten, die Registrierung auf Aktualisierungen sowie den Abruf von Daten aus dem Kontextmodell. Zudem können über den *Management Service* Domänenprofile registriert und verwaltet werden.
Der Update Service bietet u. a. die Methoden boolean add/removeContext (String provider, ContextFact() facts) zum Einfügen und Entfernen von Kontextdaten aus dem Modell. Ein ContextFact enthält u. a. das betroffene RDF-Tripel, Informationen zur Vertrauenswürdigkeit der Daten (vgl. *Confidence*-Wert in Abschnitt 6.3.1.2), und den Zeitpunkt der Erfassung.
Der Query Service bietet Methoden für die synchrone Abfrage sowie die Registrierung für asynchrone Benachrichtigungen. In beiden Fällen wird SPARQL zur Adressierung des Kontextdatums genutzt. Die drei wichtigsten Methoden umfassen:

String queryContext (String sparqlQuery)
: ... ermöglicht die Anfrage nach den im SPARQL-Query adressierten Daten des Kontextmodells.

String requestNotification (String callback , String targetNs , String sparqlQuery)
: ... dient der Registrierung von Clients bzw. *Context Consumern* – identifiziert über die Endpoint-Referenz $callback$ und den Namensraum $targetNs$ – für asynchrone Benachrichtigungen bei Änderungen der im SPARQL-Query adressierten Kontextdaten.

void removeNotificationRequest (String callback , String sparqlQuery)
: ... entfernt eine bestehende Registrierung zur asynchronen Benachrichtigung.

```
1  <soap:envelope xmlns:soap="http://.../soap/envelope/" xmlns:web="http://... /croco/">
2   <soap:header/>
3    <soap:body>
4     <web:queryContext>
5      <web:sparqlQuery>
6       PREFIX usr: < http://.../crocoon/context-user.owl >
7       SELECT  ?x
8       WHERE { ?x a usr:Email.
9               ?y a usr:UserContextProfile.
10              ?y usr:accountName "Batman".
11              ?y usr:email ?x
12       }
13     </web:sparqlQuery>
14    </web:queryContext>
15   </soap:body>
16  </soap:envelope>
```

Lst. 7.7: Synchrone Abfrage von Kontextdaten über SOAP

Codebeispiel 7.7 veranschaulicht eine einfache Anfrage über queryContext (die Namensräume wurden der Übersichtlichkeit halber verkürzt). Im *Body* der SOAP-Anfrage ist die Anfrage zu sehen, deren wichtigster Teil im `WHERE`-Abschnitt enthalten ist. Im Beispiel werden alle Konzepte vom Typ Email abgefragt, die Teil des Nutzerprofiles von „Batman“ sind: Dazu werden zunächst alle Variablen x vom Typ Email adressiert (Zeile 8 – „a“ ist die Kurzform des RDF-Prädikats *type*). Dann wird ein Nutzerprofil gesucht, dessen Prädikat accountName mit dem Wert „Batman“ belegt ist (Zeile 9–10). Diesem Profil soll die gesuchte Email zugeordnet sein.

Die Rückgabe zu dieser Anfrage erfolgt im standardisierten SPARQL Query Results XML Format [@Beckett und Broekstra, 2008].

Zur asynchronen Kommunikation mit CroCo wurde weiterhin eine auf *Comet* beruhende Schnittstelle umgesetzt. Bei Comet handelt es sich um ein Architekturprinzip, welches *Push*-Benachrichtigungen durch den Server über langlebige HTTP-Verbindungen erlaubt [Crane und McCarthy, 2008]. Sie werden von CroCo zum Versenden der Aktualisierungsnachrichten genutzt. Im Prototyp kommt dazu die freie Referenzimplementierung des Bayeux-Protokolls [@Russell et al., 2007] aus dem CometD-Projekt [@*CometD* 2012] zum Einsatz.

Der Datentransfer über Comet findet nach den Publish/Subscribe-Muster über benannte Kanäle statt. Tabelle 7.1 gibt einen Überblick über die unterstützten Kanäle. Während die obige Registrierung über SOAP im Normalfall serverseitigen MREs vorbehalten bleibt, die eine Service-Endpoint als Rückrufpunkt spezifizieren, ist die Comet-Schnittstelle auf leichtgewichtige Clients ausgelegt. Die Wahl bei der Serialisierung der Nutzdaten fiel deshalb auf JSON.

Kanal	Beschreibung	Format
/croco/subscribe	Registrierung eines Clients auf Kontextänderungen bezüglich einer SPARQL-Query	`{ query: String }`
/croco/unsubscribe	Deregistrierung des Clients für die angegebene SPARQL-Query	`{ query: String }`
/croco/unsubscribeAll	Vollständige Deregistrierung des Clients	`{}`
/croco/notify	Asynchrone Benachrichtigungen von CroCo, bestehend aus betroffener Query und SPARQL-Result	`{ query: String , result: String }`

Tab. 7.1: Kanäle zur asynchronen Kommunikation mit CroCo

Für die Umsetzung der Kontextontologie *CroCoOn* wurde in Anlehnung an SOUPA [Chen et al., 2004] ein modularer Ansatz gewählt, der sowohl der Übersichtlichkeit als auch der Performanz zuträglich ist, da bei der Verarbeitung nicht benötigte Teilontologie ausgeblendet werden können. Jeder Kontextaspekt wurde als separate Ontologie in OWL DL umgesetzt, wobei die *Upper Ontology* die gemeinsame Schnittstelle darstellt. Abbildung 7.7 zeigt schematisch die geschaffenen Ontologien und die bestehenden Import-Relationen. Für die realisierten Anwendungen wurden sie domänenspezifisch erweitert, soweit dies für die Auswahl im Integrationsprozess und die Umsetzung von Adaptionstechniken nötig war. Einige der dafür geschaffenen Konzepte sind in Abbildung 6.10 angedeutet. Dazu zählen u. a. ein Nutzerprofil, Hard- und Softwareeigenschaften, situative Kenngrößen, wie der Ort der Nutzung, und Vorlieben, z. B. hinsichtlich des Preislimits für Komponenten.

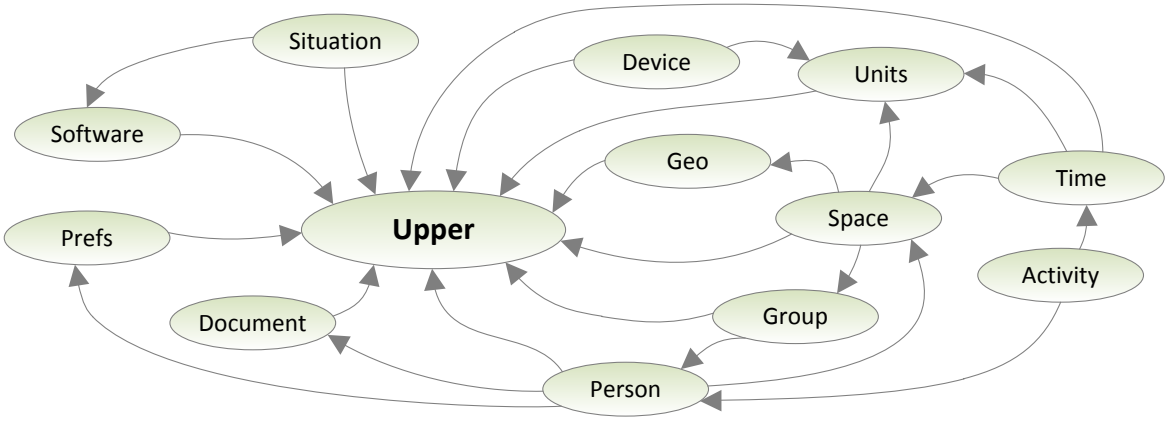

Abb. 7.7: Import-Relationen zwischen den Teilmodellen der CroCo Ontology

Über die Anwendung im Rahmen dieser Arbeit hinaus konnten die Konzepte des Kontextdienstes anhand weiterer Anwendungsszenarien, z. B. zur kontextsensitiven, semantischen Verwaltung multimedialer Dokumente, validiert werden. Entsprechende Informationen bieten Mitschick et al. (2010).

7.2.3.2 Clientseitiges Adaptionssystem

Zur Validierung der in Abschnitt 6.3 vorgestellten Adaptionskonzepte für komposite Anwendungen erfolgte die Umsetzung und Kopplung des beschriebenen Adaptionssystems mit der ausgereiftesten MRE – der TSR. Die Realisierung als eigenständiger Web Service erwies sich für die Nutzung mit der clientseitigen Laufzeitumgebung hinsichtlich der Performanz und Praktikabilität als nicht sinnvoll. Grundlage für die Implementierung bildete deshalb wiederum das JavaScript-Framework Ext Core [@Sencha, 2012]. Alle vorgestellten Module des Adaptionssystems wurden als eigenständige Klassen im Namensraum Ext.cruise.client.adapt umgesetzt.
Die folgenden Abschnitte geben einen Überblick über die Realisierung der vorgestellten Konzepte zur Kontextüberwachung und Auswertung der modellierten Adaptionsbelange. Eine detaillierte Darstellung kann Pietschmann et al. (2011a) und Radeck (2010) entnommen werden.

Lokale Kontextüberwachung und -verwaltung

Zur Überwachung der für die Beispiel- und Testanwendungen nötigen Kontexteigenschaften wurden verschiedene Kontextmonitore umgesetzt. Sie genügen der gemeinsamen Schnittstelle IContextMonitor, über deren obligatorische Methode sendContext erfasste Kontextdaten zum Context Manager gesendet werden. Durch die Implementierung dieser Schnittstelle können beliebige neue Monitore realisiert und direkt beim Context Manager registriert werden. Die wichtigsten, bislang geschaffenen Monitore umfassen:

BrowserMonitor ... erfasst Informationen über den genutzten Webbrowser, wie dessen Name, Version, verfügbare Plug-ins und unterstützte DOM-Operationen. Die Umsetzung stützt sich u. a. auf die jQuery-Bibliothek [@jQuery, 2012].

WindowSizeMonitor ... überwacht die Größe des Browserfensters.

LocationMonitor ... ermittelt über die GeoLocation API [@Popescu, 2010] die Position des Nutzers und führt über Google Maps [@Google, 2011b] ein Reverse Geocoding aus, um den Ort der Nutzung – wie im Modell gesehen, stellt die UserLocation eine Spezialisierung von City dar – abzuleiten.

BasicSystemMonitor ... liest Systemeigenschaften aus, die über JavaScript messbar sind, u. a. zu Zeitzone, Auflösung, Farbtiefe, Plattformbeschreibung und Betriebssystem.

DeviceMonitor ... ermittelt über WURFL [@WURFL, 2012] anhand des UserAgent-Headers möglichst genaue Informationen zum Endgerät des Nutzers.

Das lokale Kontextmodell, d. h. die Menge der gebundenen Eigenschaften des Kontextobjektes aus dem Kompositionsmodell, wird durch die Klasse AdaptationContext repräsentiert und vom Context Manager verwaltet. Seine Einbindung in den globalen Datenfluss erfolgt, wie für andere Mashup-Komponenten auch, über den Event Broker. Dazu erhält es einen dedizierten Kanal, über den es Kontextänderungen in Form von Ereignissen anzeigen kann.

Für die Anbindung des Kontextdienstes CroCo wurde ein entsprechender CSA – der CroCoAdapter – realisiert. Er übernimmt die Konvertierung der ausgetauschten Daten von der Repräsentation als JSON-Tupel in RDF/XML. Auch die Auswertung komplexer SPARQL-Anfragen, wie sie in den Bedingungen von Adaptionsaspekten formuliert sein können, wird an CroCo über dessen synchrone SOAP-Schnittstelle weitergeleitet. Für alle benötigten Kontextdaten, z. B. als Trigger für Adaptionen, erfolgt bei der Initialisierung des Adapters durch den Adaptation Manager die Registrierung auf asynchrone Änderungsnachrichten über die beschriebenen Comet-Kanäle. Die dafür entwickelte Klasse CroCoComet greift zur Kommunikation, wie CroCo auch, auf Funktionalitäten des CometD-Frameworks [@*CometD* 2012] zurück.
Wie bereits angedeutet, obliegt es dem Adapter, aus den Adressierungspfaden im Kompositionsmodell bzw. der Kontextkomponente die korrekten SPARQL-Queries für Abruf und Registrierung zu generieren. Dazu wird für jeden Teilpfad ein Tripel angelegt und die Variablen entsprechend der Properties im Modell benannt, um eine korrekte Zuordnung zu ermöglichen. Noch komplexer stellt sich die Erzeugung von RDF-Tripeln zur Aktualisierung von Kontextdaten dar. Die Monitore sind so umgesetzt, dass sie die überwachten Kontextdaten anhand eines eindeutigen Adressierungspfades im Kontextmodell ausweisen. Die Einbindung der Daten ist für generische Informationen unkritisch. Sie erfordert aber mehr Aufwand bei nutzerspezifischen Informationen, da diese zunächst dem jeweiligen Benutzerprofil (UserProfile, vgl. Abbildung 6.10) zugeordnet werden müssen.
Ein zusätzliches Problem ist hierbei das Fehlen von Instanzdaten im Adressierungspfad. Angenommen, der Ladezustand des Geräteakkus ist im Kontextmodell aus Abbildung 6.10 als Eigenschaft batteryLevel von DeviceState modelliert. Sendet ein Monitor nun entsprechende Daten, müssen diese dem jeweils richtigen Nutzerprofil über ComputationalDevice und DeviceState zugeordnet werden. Falls Instanzen zum Aufbau dieser Beziehung fehlen, müssen zunächst RDF-Tripel, z. B. für DeviceState, angelegt werden, bevor die Eigenschaft batteryLevel auf den gemessenen Wert gesetzt werden kann.
Um dies zu ermöglichen, verfügt der Adapter über Informationen zur Kontextmodellstruktur. Um diese von CroCo zu erhalten, wurde letzteres um eine Methode erweitert,

die Strukturinformationen – in diesem Fall als JSON-Serialisierung – bereitstellt. Der CroCoAdapter wird somit in die Lage versetzt, vor der Modellaktualisierung über SPARQL-Queries das Vorhandensein aller nötigen Knoten sicherzustellen und diese ggf. selbst anzulegen, bevor er neue Daten hinzufügt. Aufgrund der umfangreichen Ausdrucksmöglichkeiten der Kontextontologie und der Generik von CroCo gestaltet sich das Hinzufügen neuer Daten dennoch sehr aufwändig. Daher ist die prototypische Implementierung des Adapters in ihrer Komplexität limitiert, z. B. was die Überprüfung von Kardinalitäten betrifft. Eine spätere Erweiterung ist jedoch problemlos möglich.

Umsetzung von Context Links

Die Initialisierung des Adaptionssystems erfolgt über dessen Stellvertreter und öffentliche Schnittstelle – den Adaptation Manager. Durch die gleichnamige Klasse werden alle in Abschnitt 6.3 vorgestellten Module verwaltet und initialisiert, die jeweils als eigenständige Klassen umgesetzt wurden. Da der Manager Zugriff auf das Kompositionsmodell erhält, kann er die relevanten Teile an die Module des Adaptionssystems weiterleiten.

Durch den Context Manager werden u. a. die formulierten Context Links ausgewertet. Bei denjenigen, die über das Attribut `isInitial` markiert sind und nur zur initialen Belegung von Eigenschaften benötigt werden, registriert sich der Adaptation Manager auf das *Life-Cycle-Event*, das die Instanziierung der angeschlossenen Komponenten anzeigt. Wird es ausgelöst, ruft er den betroffenen Kontextparameter vom AdaptationContext ab und setzt ihn durch Aufruf von setProperty für die Komponente, bevor diese den Zustand *Integrated* erreicht.

Alle sonstigen, permanenten Context Links werden auf Links abgebildet, die beim Event Broker zwischen *Change*-Events vom AdaptationContext und den Setter-Methoden der Komponenteneigenschaften angelegt werden. Die Vermittlung erfolgt dann durch den Broker, wie bereits beschrieben. Das Context-Link-Attribut `threshold` wird auf die Eigenschaft *syncThreshold* der Kommunikationskanäle abgebildet, welches auch bei der Interpretation der BackLinks genutzt wurde. Beim Versenden von Daten prüft der Broker dann, ob das minimale Zeitintervall zwischen zwei Aktualisierungen eingehalten wird, ansonsten wird die Nachricht verworfen und resultiert in der Rückgabe einer entsprechenden Fehlernachricht (vgl. Statuscode, Abschnitt 6.2.4).

Jeder permanente Context Link wird allerdings auch initial synchronisiert. Somit tritt die gewünschte Wirkung sofort in Erscheinung, auch für Kontextdaten, die sich selten ändern. Bezieht sich ein Link z. B. auf den Ort des Nutzers, so wird dieser direkt bei der Initialisierung mit den angeschlossenen Properties synchronisiert, ohne die erste Aktualisierung abzuwarten.

Interpretation und Auswertung von Adaptionsaspekten

Die Beschreibung des adaptiven Verhaltens liegt beim Start einer Anwendung in Form von Adaptionsaspekten im Adaptivity Model vor (vgl. Abschnitt 5.3.2). Mit der Initialisierung des Adaptionssystems wird dieses durch die Klasse RuleEngine interpretiert. Die formulierten Adaptionsaspekte werden dabei zu ECA-Regeln in Form von JSON umgewandelt, die im Weiteren durch die *Rule Engine* verwaltet und im Bedarfsfall evaluiert werden.

Falls bei der Formulierung auf semantische Templates zurückgegriffen wurde, werden diese durch die Rule Engine aufgelöst, die für jedes der abstrakt adressierten Ziele eine spezifische ECA-Regel erstellt. Die hierarchisch geschachtelten Bedingungen werden durch den rekursiven Aufruf der privaten Methode _buildCondition zu einer JavaScript-Anweisung aggregiert, in der die Operatoren und Kontextparameter durch Hilfsfunktionen repräsentiert werden. Verweise auf Kontextdaten – im Modell Referenzen auf Properties von Context – werden beispielsweise zu Aufrufen an die Klasse AdaptationContext, und zur Auswertung von Event-Parametern wird auf die Message-API des Event Brokers zurückgegriffen. Schließlich werden alle Regeln nach der Priorität der zugrunde liegenden Aspekte sortiert und die Rule Engine registriert sich beim Event Broker auf die darin referenzierten Trigger-Events.
Sobald diese eintreffen, werden sie in der Methode handleEvent verarbeitet. Codebeispiel 7.8 zeigt einen Ausschnitt der Methode, die eintreffende Kontextereignisse mit den Triggern der Regeln vergleicht. Wie zu sehen ist, kommt bislang lediglich ein sequentieller Algorithmus zum Einsatz. In Zeile 4 wird zunächst ein Array für die auszulösenden Regeln angelegt. Danach wird über alle Regeln (Zeile 5) und ihre Trigger-Ereignisse (Zeile 8) iteriert, die jeweils mit dem eingetroffenen Ereignis verglichen werden. Im Fall der Übereinstimmung werden potentiell vorhandene Bedingungen geprüft (Zeile 16). Die Auswertung des über _buildCondition erstellten JavaScript-Ausdrucks erfolgt in _validateCondition mittels *eval*, da die native Prüfung in JavaScript eine deutlich effizientere und schnellere Verarbeitung verspricht, als die Interpretation einer extra geschaffenen Datenstruktur. Schließlich werden die anzuwendenden Regeln der erstellten Liste hinzugefügt (Zeile 17/19).

```
1  handleEvent: function( message, channel ){
2    // Auslesen des Ereignisnamens (eventName) und der Daten (eventData)
3    ...
4    var triggering= new Array();    // Array ausgelöster Regeln
5    for (var i=0; i < this.rules.length; ++i) {    // Suche nach passenden Regeln
6      var rule = this.rules[i];
7      var triggerEvents = rule.triggerEvents;
8      for (var te=0; te < triggerEvents.length; ++te) {
9        var currEvent = triggerEvents[te];
10       // Prüfe Regelauslösung durch (reserviertes) Kontextereignis
11       if (eventName == "_contextChanged") {
12         // Vergleich von übermitteltem und Trigger-Parameter der Regel
13         if (currEvent.type != eventData.parameter[1].type)
14           continue;  // Abbruch bei Ungleichheit
15         if (rule.condition) {    // ECA-Regel
16           if (this._validateCondition(rule.condition.js, eventData)==true)
17             triggering.push(rule);
18         } else {                 // EA-Regel
19           triggering.push(rule);
20         }
21       }
22       ...  // Prüfe sonstige Ereignisse
23   }}
24   // Sortierung der Regeln nach ihrer Priorität
25   triggering = this._sort(triggering);
26   // Ausführen der Regeln
27   this._executeRules(triggering);
28 }
```

Lst. 7.8: Dynamische Bestimmung ausgelöster Adaptionsregeln

Eine Sonderbehandlung erfahren die Regeln, deren Bedingung oder Aktion auf den Wert des auslösenden Kontextereignisses Bezug nehmen. Sie werden ohne Trigger

direkt beim Anwendungsstart einmal ausgeführt. Hintergrund sind Kontextinformationen, die sich u. U. nicht häufig ändern und somit erst zu spät zur Anpassung führen würden. Bezieht sich beispielsweise ein Aspekt auf den Ladezustand des Endgerätes, so sollte er direkt bei der Initialisierung der Anwendung mit den vorliegenden Kontextwerten ausgelöst werden, ohne auf die erste Veränderung bzw. Aktualisierung warten zu müssen.
Die bereits beim Parsen nach ihrer Priorität sortierten Adaptionsaktionen jeder auslösenden Regel werden anschließend an den Adaptation Manager zur Ausführung übergeben.

Umsetzung von Adaptionsaktionen

Für die in 5.3.1 beschriebenen Adaptionstechniken bzw. die daraus abgeleiteten *Advices* aus der Aspektbeschreibung (Abschnitt 5.3.2.2) wurde jeweils eine Implementierung der Schnittstelle IAction erstellt. Letztere stellt den plattformspezifischen Implementierungen Referenzen auf alle nötigen Module der MRE zur Verfügung, u. a. Adaptation, Component und Context Manager. Weiterhin definiert sie u. a. die zentrale Methode execute, die der Ausführung der jeweiligen Adaptionen dient.

Auszuführende Adaptionsaktionen werden nach der Regelauswertung zunächst beim Adaptation Manager in einer Warteschlange eingereiht und nach dem FIFO-Prinzip sequentiell abgearbeitet. Grundlage für diese bereits in Abschnitt 6.3.3 erwähnte Adaptionssperre bildet die Klasse DelayedTask des zugrunde liegenden Ext-Frameworks. Die Ausführung einer Aktion wird schließlich durch den Aufruf der Methode execute ausgelöst. Sie kann prinzipiell asynchron zur restlichen Anwendung verlaufen, sodass die Freigabe der Adaptionssperre mit dem Aufruf der Methode _actionReady durch die abgeschlossene Aktion selbst erfolgen muss.
Die clientseitig umgesetzten Aktionen umfassen ChangeLayout, RemoveComponent, ReconfigureComponent, ExchangeComponent, SetVisibility, RemoveSubscriber, AddSubscriber, FireEvent, AddPublisher und RemovePublisher. Sie setzen somit alle identifizierten Adaptionstechniken um, mit Ausnahme der Screenflow-Anpassung.
Die Komplexität der Aktionen unterscheidet sich naturgemäß erheblich. Während häufig nur eine Delegation an bestehende Module der MRE nötig ist – z. B. bei der Änderung des Layouts – sind andere Aktionen deutlich aufwändiger. Hierzu zählt der Komponententausch, bei dem das in Abschnitt 6.3.3 vorgestellte Phasen-Commit-Protokoll und der Zustandstransfer realisiert werden mussten.
Aus Platzgründen wird an dieser Stelle auf die ausführliche Darstellung der Realisierung aller Techniken verzichtet. Das folgende Beispiel soll jedoch einen praktischen Einblick in zumindest eine Aktion geben. Codebeispiel 7.9 zeigt einen Ausschnitt der Klasse ReconfigureComponent, die der Änderung einer Komponenteneigenschaft dient. In Zeile 1 ist die Erweiterung der Schnittstelle IAction erkennbar, durch die der Zugriff auf die MRE-Module gegeben ist (Zeile 4). In Zeile 8 beginnt die Methode zur Ausführung der Adaption, der aus der Regelbeschreibung alle nötigen Informationen, wie Name, Wert und Pointcuts, zur Verfügung stehen (Zeile 9-11). Für jeden Pointcut erfolgt die Auflösung über den Component Manager (Zeile 14) und schließlich das Setzen der neuen Belegung (Zeile 19). In diesem Beispiel ausgespart ist die Auflösung möglicher Referenzen auf Kontextparameter, die über den Context Manager erfolgt.

```
1  Ext.cruise.client.adapt.impl.ReconfigureComponent =
2      Ext.extend(Ext.cruise.client.adapt.IAction, {
3
4    constructor: function(adaptMan, componentMan, contextMan, logger){
5      ...
6    },
7
8    execute: function(action){
9      var pname = action.config.property;
10     var pvalue =action.config.value;
11     var pointcut = action.pointcut;
12     // Iteration über alle definierten Pointcuts
13     for (var idx=0; idx < pointcut.length; ++idx){
14         var comp = this.componentMan.getComponentInstance(pointcut[idx]);
15         if (!comp) continue; // Abbruch, falls Pointcut nicht gefunden wurde
16         // Auflösen von Kontextreferenzen über den Context Manager
17         ...
18         // Rekonfiguration
19         comp.setProperty(pname, pvalue);
20     }
21     // Rückruf zum Auflösen der Adaptionssperre
22     this.adaptMan._actionReady(action);
23   }
24 });
```

Lst. 7.9: Dynamische Bestimmung ausgelöster Adaptionsregeln

Adaptierbarkeit

Durch Erweiterungen der TSR wurde weiterhin die Möglichkeit geschaffen, einige der über das Adaptionssystem bereitgestellten Aktionen durch Nutzer manuell auszulösen. Dazu wird jedes Panel, das eine Komponente umgibt, mit Schaltflächen erweitert, die u. a. die Minimierung, Maximierung und Entfernung der entsprechenden Komponente erlauben. Zudem ist es möglich, Komponenten durch Alternativen zu ersetzen, indem sie von der TSR als Templates aufgefasst werden. Nutzer können hierzu ein Drop-Down-Feld nutzen, wie es in Abbildung 7.11 in der Kopfzeile der Bildergalerie zu sehen ist. Mit der Auswahl eines alternativen Kandidaten wird die o. g. Aktion zum Komponententausch angestoßen.

Mit der vollständigen Umsetzung des in Abschnitt 6.3 vorgestellten Adaptionssystems wurde der Grundstein für die Adaption kompositer Anwendungen gelegt. Es konnte gezeigt werden, dass die Beschreibung von Adaptionslogik plattformunabhängig als Teil des Kompositionsmodells erfolgen, und zur Laufzeit auf eine bestehende, universelle Komposition angewendet werden kann. Die Kopplung mit dem Kompositionssystem wurde am Beispiel der TSR demonstriert, die um entsprechende Adaptionsaktionen erweitert wurde.
Die in Kapitel 6 vorgestellten, neuen Konzepte zur universellen Komposition von Webanwendungen konnten somit vollständig umgesetzt werden. Neben der grundlegenden Infrastruktur zur Verwaltung von Komponenten und Modellen wurden sowohl der gesamte, semantische Integrationsprozess realisiert als auch die Komposition und Ausführung von Anwendungen auf verschiedenen Plattformen erprobt. Anhand einer ausgewählten MRE konnte schließlich die Validität der konzipierten Adaptionskonzepte nachgewiesen werden. Neben rein funktionalen bzw. Unit-Tests der Einzelbestandteile wurde das neuartige Gesamtkonzept von der Modellierung bis zur Ausführung an einer Reihe von Beispielanwendungen erprobt, deren Vorstellung sich der nächste Abschnitt widmet.

7.3 Validierung und Diskussion anhand der Beispielszenarien

Zur Validierung des Konzeptes, zu Test- und Demonstrationszwecken wurden u. a. im Rahmen diverser studentischer Arbeiten und Praktika eine Reihe prototypischer Anwendungen entwickelt, die mit den in Abschnitt 2.2 vorgestellten Szenarien korrespondieren. Sie wurden auf Basis der in den letzten Abschnitten dargestellten Referenzimplementierung umgesetzt und hinsichtlich der in Abschnitt 2.3 formulierten Anforderungen evaluiert.
Das Ziel der praktischen Realisierungen bestand nicht darin, den Mehrwert der Anwendungen gegenüber konventionellen Lösungen zu belegen. Vielmehr sollte gezeigt werden, dass die Modellierung interaktiver, kompositer Anwendungen auf Basis des vorgestellten Kompositionsmodells plattformunabhängig und in ausreichender Mächtigkeit erfolgen kann, um den o. g. Anforderungen gerecht zu werden. Zudem sollten die Laufzeitkonzepte der Discovery, Integration und Ausführung sowie die Adaptionskonzepte auf ihre Belastbarkeit überprüft werden.

Grundlage hierfür bildeten eine Vielzahl von Komponenten, die für die Anwendungen entwickelt wurden. Einige von ihnen kamen auch plattformübergreifend zum Einsatz. Abbildung 7.8 zeigt eine Zusammenstellung einiger der umgesetzten UI-Komponenten. Sie umfassen u. a. Karten (Google Map, OpenStreetMap), verschiedene Diagrammtypen (Google Visualization API), eine Bildergalerie, einen YouTube-Player, einen Feed-Viewer, eine Tag Cloud und eine HeatMap. Technologisch reichen die realisierten Komponenten von einfachem HTML und JavaScript bis zu Flash, Silverlight und Google GWT. Bei der Umsetzung wurde insbesondere auf die Unabhängigkeit von den Datenquellen geachtet. Die Schnittstellen sind dementsprechend generisch, sodass die Verknüpfung mit verschiedenen Service-Komponenten je nach Anwendungskontext möglich wird.

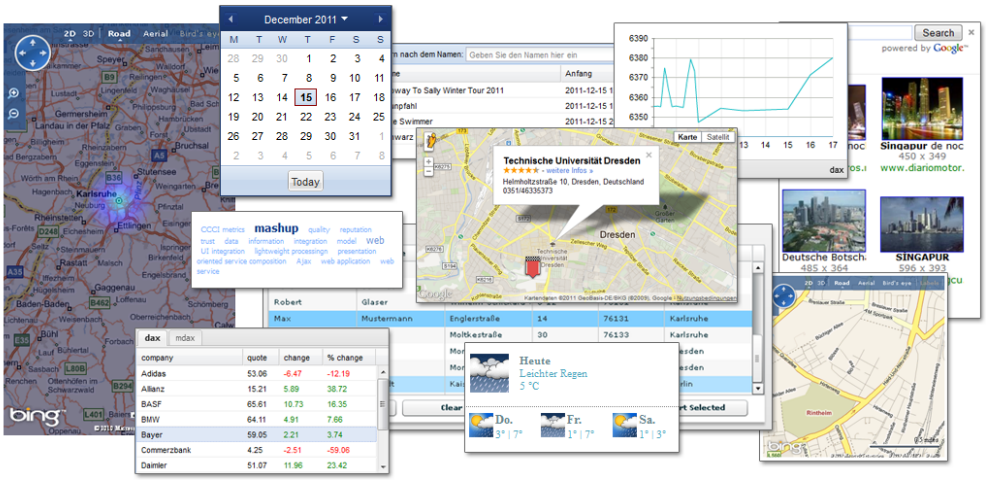

Abb. 7.8: Auswahl einiger der umgesetzten UI-Komponenten

Mit Hilfe dieser Komponenten wurden u. a. drei Beispielanwendungen umgesetzt: Zum Nachweis der grundlegenden Funktionalität wurde die Anwendung *TravelMash* realisiert. Die erweiterten Kommunikations- und Koordinationsmuster sowie die Unterstützung von *Drag-and-Drop* wurden mit der Komposition *StockMash* verifiziert. Schließlich diente das Mashup *TravelGuide* der Validierung der Adaptionskonzepte. Auf alle drei Anwendungen wird im Folgenden kurz eingegangen.

7.3.1 Reiseplanung mit *TravelMash*

Zur Validierung der Basiskonzepte sowie der prototypischen Implementierung der Kompositionsinfrastruktur wurde als Teil des umgebenden Forschungsprojektes [@CRUISE, 2012] zunächst das in Abschnitt 2.2.1 vorgestellte Reiseplanungsszenario als Anwendung umgesetzt. So konnten u. a. die Ausdrucksstärke des Modells, deren Interpretation, die dynamische Suche und Integration von Komponenten, die ereignisorientierten Kommunikation über Links und die Umsetzung der Layout- und Styling-Konzepte verifiziert werden.

Wie im Szenario beschrieben, ermöglicht *TravelMash* die Planung einer Reise zu einem kulturellen Ereignis und kombiniert dazu UI- und Service-Komponenten, die Ereignisse, Hotelinformationen und Routeninformationen liefern, visualisieren und in Beziehung setzen. Als grundlegende Datenlieferanten wurden die Dienste von Last.fm [@LAST.FM, 2012] (Konzerte), Kayak [@KAYAK, 2012] (Hotels) und dem Verkehrsverbund Oberelbe (VVO) [@VVO-NAVIGATOR, 2012] (Routenplanung) genutzt und als clientseitige Service-Komponenten umgesetzt. Zudem wurden die in Abbildung 2.2 sichtbaren UI-Komponenten realisiert. Dabei wurde größtenteils auf das bereits angesprochene JavaScript-Framework Ext JS zurückgegriffen, aber auch Google Maps und Google Weather fanden Verwendung.

Schließlich wurden alle Komponenten mit einer SMCDL beschrieben und im CoRe registriert. Das Codebeispiel in Anhang A.1 zeigt auszugsweise die Beschreibung der Komponente *RoutingMap*, wie sie in TravelMash zum Einsatz kommt. Grundlage für die Typisierung der Properties und Parameter bildeten semantische Konzepte, die für die Reiseplanungsdomäne auf Basis von OWL DL in Ontologien formalisiert wurden. Für die funktionalen Annotationen wurde auf die Formalisierungen von TIETZ et al. (2011) zurückgegriffen, die der Repräsentation durchzuführender Aufgaben und Ereignisse aus der gleichen Domäne dienen.

Die Modellierung von TravelMash erfolgte auf Basis des vorgestellten Kompositionsmetamodells mit Hilfe des o. g. Baumeditors. Ein Ausschnitt der Modellserialisierung war bereits in Codebeispiel 7.1 zu sehen. Eine ausführlichere Darstellung kann Anhang A.3 entnommen werden.

Abbildung 7.9 zeigt Teile des Modells in der Baumdarstellung des grafischen Editors. Auf der linken Seite (A) ist das Conceptual Model aufgeklappt, welches im unteren Teil die Komponenten der Anwendung aufweist. Unter den drei Service-Komponenten zur Bereitstellung von Hotels, Konzerten und der Routenplanung, ist die UI-Komponente zur Darstellung von Konzerten erweitert. Es ist erkennbar, dass sie über eine Operation *updateItems* zur Aktualisierung der angezeigten Werte, über Ereignisse zur Ausgabe des jeweils aktuellen Ortes *currentLocation* sowie des selektierten Konzertes *itemSelected* und einige Eigenschaften verfügt. Beispielhaft ist darunter eine funktionale Annotation – die Fähigkeit (*Capability*) zur Sortierung – zu sehen.

Auf der rechten Seite (B) ist im oberen Bereich das Layout-Modell mit zwei absolut definierten Layouts erkennbar. Für die breite Variante *wide* sind hier die Koordination (gekürzt) mit zugewiesenen UI-Komponenten aufgelistet. Darunter ist das Kommunikationsmodell zu sehen. Über den ersten PropertyLink *Destination* erfolgt die Synchronisation des Zielortes zwischen allen Komponenten (gekürzt). Der folgende BackLink *Load Events* verknüpft das gerade erwähnte Ereignis *currentLocation* der Konzert-Anzeige mit dem Konzert-Dienst, sodass der Wechsel

des aktuellen Ortes zum Abruf und schließlich zur Aktualisierung der Ereignisse in der Benutzeroberfläche führen. Der Link *SelectedHotel* dient der unidirektionalen Weitergabe eines ausgewählten Hotels an die Komponente *Start-Destination*.
Letztere erhält ihre Benennung bewusst, da sie als Template markiert ist und nicht zwingend eine Karte darstellen muss. Funktionale Anforderungen bestehen lediglich in der Auswahl von Start- und Zielort, sodass für mobile Geräte mit begrenzter Ausgabefläche stattdessen ein Formular zur Texteingabe und zur Auswahl vordefinierter Orte integriert wird.

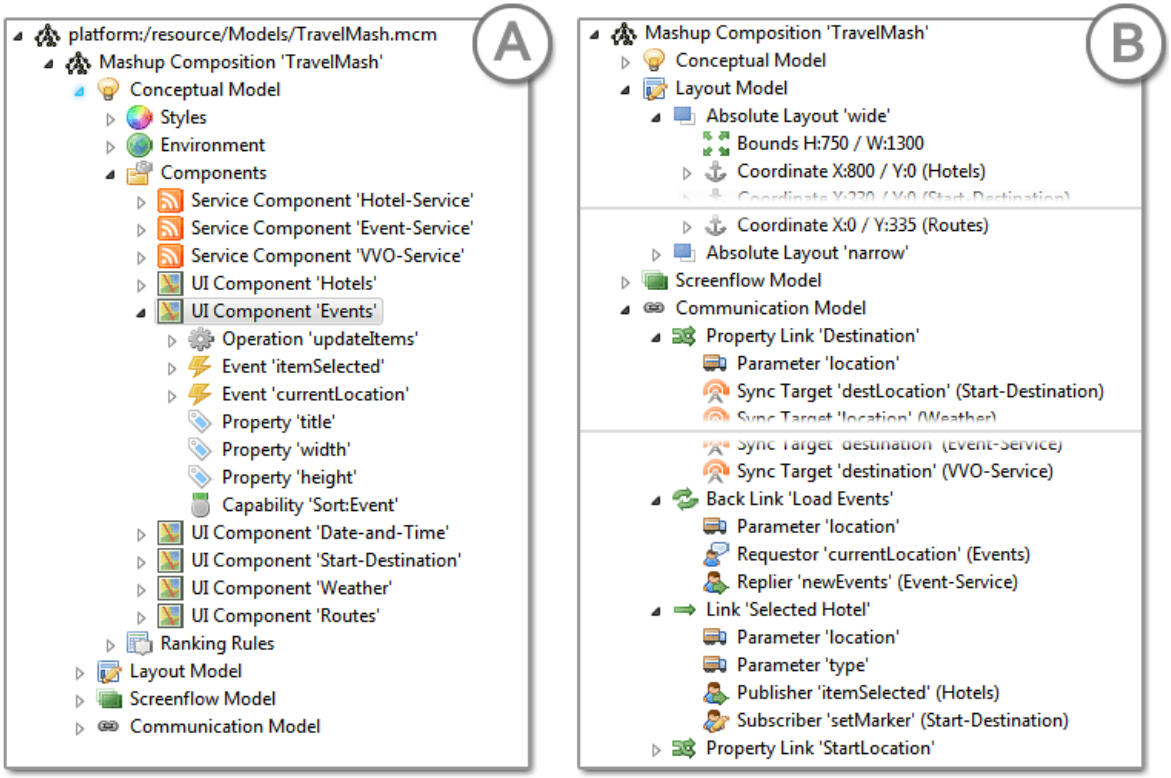

Abb. 7.9: Ausschnitte aus der grafischen Modellrepräsentation von *TravelMash*

Neben der ausreichenden Mächtigkeit des Metamodells könnten an TravelMash schließlich die inhärenten OCL-Constraints und die in Abschnitt 7.1.3 beschriebenen Generierungsvorschriften validiert werden. Sie erlaubten das Deployment in eine ausführbare, clientseitigen Webanwendung. Codebeispiel 7.10 zeigt mit einigen Vereinfachungen die resultierende XHTML-Seite, die die Initialisierung der TSR anstößt und zur Interpretation des darin referenzierten Kompositionsmodells führt. Neben den üblichen Header-Informationen ist in Zeile 7 das Laden der TSR-Skripte zu sehen. Das Skript ab Zeile 8 der Initialisierung von TSR und Anwendung durch die Methode initApp in Zeile 11. Diese wird aufgerufen, sobald das Laden aller JavaScript-Dateien der TSR durch die Methode loadRuntime in Zeile 25 vollständig abgeschlossen ist. Bei der Initialisierung wird in Zeile 14 zunächst der Application Manager mit den Uniform Resource Locators (URLs) des Kompositionsmodells und der nötigen Infrastrukturdienste instanziiert. Der Manager übernimmt danach selbständig die Instanziierung der restlichen Module. Im Fehlerfall erfolgt im Beispiel eine Log-Ausgabe (Zeile 20).

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" ...>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
  <title>TravelMash | VVO Verbindungsauskunft</title>
  <script type="text/javascript" src="ThinServerRuntime.js"></script>
  <script type="text/javascript">
    var applicationManagerInstance;
    // Callback-Methode zur Initialisierung von TSR und Anwendung
    initApp = function() {
      Ext.onReady( function() {
        try {
          applicationManagerInstance = new Ext.cruise.client.ApplicationManager({
              modelURL: "http://... /TravelMash.ccm",
              coreURL: "http://... /CoReService",
              msURL: "http://... /MediationService"
          });
        } catch(e){
            log4javascript.getDefaultLogger().fatal('error@InitPage' , e);
        }
      });
    }
    // Laden der TSR und Rückruf von 'initApp'
    loadRuntime(initApp);
  </script>
</head>
<body/>
</html>
```

Lst. 7.10: Startseite zum Laden einer TSR-basierten Mashup-Anwendung

Die Komposition und Ausführbarkeit von TravelMash wurde auf Basis der TSR-Implementierung und im Zusammenspiel mit den restlichen Diensten (CoRe, TyRe, Mediation Service, etc.) erfolgreich in verschiedenen Browsern, u. a. Mozilla Firefox (ab Version 4), Google Chrome (ab Version 14) sowie Internet Explorer (ab Version 8) getestet. Die mittlere Ladezeit bis zur vollständigen Funktionsfähigkeit der Anwendung lag in Google Chrome 17 bei 3,98 s. Davon entfallen durchschnittlich 0,47 s auf die initiale Anfrage und 0,52 s auf das Laden der TSR-Skripte samt allen benötigten Bibliotheken. Die Auflösung der referenzierten Komponenten im CoRe sowie das Laden derselben schlug mit insgesamt 0,3 s zu Buche – die Anfrage nach einer SMCDL aus 64 registrierten Komponenten nahm nur 9–12 ms in Anspruch.
Der größte zeitliche Anteil entfiel auf das „Zeichnen“ bzw. Rendering der Komponenten selbst, wie weitere Tests bestätigten. Allein das Laden der Kartendaten durch Google Maps dauerte durchschnittlich 1,53 s. Ohne die Kartenkomponente konnte die Gesamtladezeit von TravelMash auf 2,4 s reduziert werden. Der immense Einfluss, der durch das Rendering von Komponenten und ihr Nachladen von Daten besteht, wird durch folgende Messreihe deutlich: Die TSR wurde so angepasst, dass die Life-Cycle-Methode *show* nie aufgerufen wird. UI-Komponenten wurden somit initialisiert (Zustand *Active*), jedoch nicht gezeichnet. Die mittlere Ladezeit von TravelMash lag unter diesen Bedingungen bei lediglich 1,33 s. Abzüglich der unabhängigen Anfragezeit für den initialen Request kann folglich für das Laden der TSR, des Kompositionsmodells und der neun enthaltenen Komponenten ein Zeitraum von 0,87 s veranschlagt werden.
Der Einfluss durch die Initialisierung der TSR, die Interpretation des plattformunabhängigen Modells und die dynamische Integration von Komponenten ist somit für

die prototypische Implementierung vertretbar gering. Er ist zudem nur marginal von der Anzahl der zu integrierenden Komponenten abhängig, wie die Anfragezeit pro Komponente von durchschnittlich 11 ms zeigt. Im Vergleich zur Gesamtladezeit fällt dieser Zeitraum bei einer üblichen Anzahl von Komponenten nicht ins Gewicht. Bei Testreihen bewegte sich die Unterschiede der Ladezeiten zwischen TravelMash mit 8, 15 und 30 Komponenten innerhalb der natürlichen Schwankungen der Anfragezeiten – das Rendering selbstverständlich ausgeschlossen.

Mit der Umsetzung von TravelMash wurden die grundlegenden Annahmen und Designentscheidungen der Konzeption bestätigt. Die Modellierung der Anwendung diente u. a. als Nachweis der Vollständigkeit und Praktikabilität des Kompositionsmetamodells. Alle funktionalen Anforderungen aus dem Szenario ließen sich auf das Modell abbilden und plattformunabhängig repräsentieren. Gleiches galt für die Realisierung der entsprechenden Komponenten und ihre Beschreibung in SMCDL. Sowohl Dienstleistungen Dritter, wie von Kayak und VVO, als auch Teile der Benutzerschnittstelle konnten einheitlich bzw. universell als Komponenten gekapselt werden. Ihre dynamische Suche und Integration zur Laufzeit untermauerten die Validität der Discovery-Konzepte und die Anwendbarkeit des Prinzips der späten Bindung auf UI-Bestandteile. Mit der vollständigen Funktionsfähigkeit der Anwendung wurden schließlich die Konzepte für Kommunikation, Koordination und Dienstzugriff auf Basis der TSR-Implementierung nachgewiesen.

7.3.2 Aktienverwaltung mit *StockMash*

Anhand des zweiten Szenarios aus Abschnitt 2.2 erfolgte die Überprüfung der erweiterten Kommunikations- und Koordinationskonzepte, wie die Umsetzung von BackLinks, PropertyLinks und der komponentenübergreifenden Drag-and-Drop-Interaktion. Die im Rahmen einer Diplomarbeit realisierte Anwendung *StockMash* dient der Analyse und Verwaltung eines Aktiendepots und wurde bereits in Abbildung 2.3 gezeigt [Wende, 2011]. Darin sind die fünf implementierten UI-Komponenten ①–⑤ zu sehen, wobei die Detailansicht ④ zweifach instanziiert wurde. Zusätzlich wurden zwei Service-Komponenten umgesetzt, die im Hintergrund die benötigten Aktienkurse und -indizes sowie Depotdaten bereitstellen.
Wie bei TravelMash wurden zur Typisierung der Komponentenschnittstellen semantische Modelle und Groundings erstellt oder wenn möglich übernommen, die in diesem Fall Konzepte der Finanzdomäne (Aktie, Aktienkurs, etc.) repräsentieren. Darauf aufbauend erfolgte die Beschreibung der Komponenten in SMCDL und die Registrierung im CoRe. Danach wurde die Anwendung mit dem angesprochenen Modell-Editor erstellt. Auf einige Besonderheiten des resultierenden Kompositionsmodells wird im Folgenden kurz eingegangen.

Die Aktienliste ① und die Verlaufsanzeige der Indizes ② sind über einen Link verbunden. Der Wechsel des Index-Tabs in der ersten Komponente führt zum Ereignis *indexChanged*, welches die Operation *setCurrentIndex* der zweiten Komponente aufruft. Der Abruf der Aktien- und Depotwerte erfolgt hingegen über BackLinks: Die Service-Komponente *StockService* besitzt dazu die Operationen *getIndexValue* und *getStockDetails*, die über entsprechende Callback-Events die Rückgabewerte bereitstellen. Die Verlaufsdarstellung der Aktienindizes ② und die Detailansichten ④ können darüber die benötigten Werte in Erfahrung bringen.

Abbildung 7.10 veranschaulicht die Verknüpfung am Beispiel der Detailansichten. Ändert sich die darzustellende Aktie, so wird zunächst das Ereignis *stockSelected* ausgelöst. Dieses ist bei beiden Instanzen über einen BackLink mit der Operation *stockOfInterest* des Aktiendienstes verbunden. Wird letztere ausgelöst, führt ihr Rückgabe-Ereignis *stockDetails* zum Aufruf der Callback-Operation *update* der jeweils aufrufenden Komponente. Die Vermittlung der korrekten Komponente erfolgt über die Callback-ID, wie in Abschnitt 6.2.4 erläutert. Der etablierte Kanal ist im Modell mit einem `syncThreshold` von 5 s versehen, wodurch die Detailansichten periodisch, jedoch nicht öfter als alle 5 s mit aktualisierten Werten versorgt werden. Zwischen den zwei Instanzen der Detailansicht besteht zudem ein PropertyLink, der für die wechselseitige Synchronisation ihrer Eigenschaften *interval* sorgt. Der Wechsel zwischen der Tages-, Monats- und Jahresansicht vollzieht sich somit immer für beide Komponenten, wodurch die Vergleichbarkeit der Werte gegeben ist.

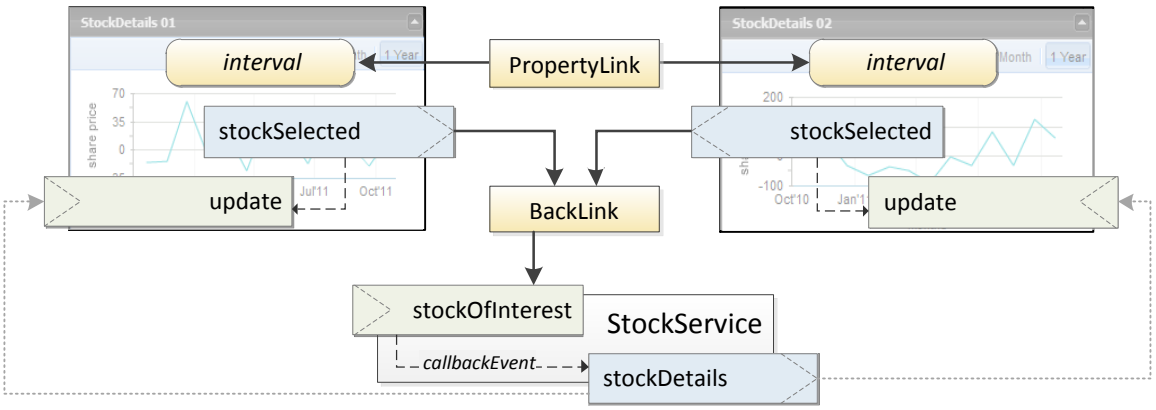

Abb. 7.10: Ausschnitt der Kommunikationsbeziehungen in *StockMash*

Die Verknüpfung von Aktienübersicht ① und Orderkomponente ③ muss hingegen nicht explizit modelliert werden. Neben einem klassischen Link kann sie – aufgrund der Übereinstimmungen von *dataSource* auf der einen und Zieloperationen auf der anderen Seite – auch über Drag-and-Drop ermöglicht werden. Gleiches gilt für die Verbindung von Aktienübersicht und Detailansichten. Gerade im letzten Fall ist ohnehin keine eindeutige Modellierung möglich: Da die Übersicht nur über ein ausgehendes Ereignis *stockSelected* verfügt, muss dieses im Bedarfsfall mit der Zieloperation und -komponente der Wahl verbunden werden. Dies geschieht in StockMash durch den Nutzer mittels Drag-and-Drop.

Mit der Realisierung von StockMash konnte gezeigt werden, dass neben den grundlegenden Integrations- und Kompositionskonzepten für universelle Mashups auch effiziente und ausreichend mächtige Kommunikationskonzepte geschaffen wurden. So kann neben der weit verbreiteten, unidirektionalen Verknüpfung von Ein- und Ausgängen auch die Zwei-Wege-Kommunikation unter Einhaltung der losen Kopplung zwischen den Komponenten ermöglicht werden.
Sieht man von der Behandlung von Sonderfällen, wie Zyklen und permanenten, asynchronen Aktualisierungen, ab, ließen sie diese Modellierungsmittel auf „einfache“ Links zurückführen. Der Vorteil der dedizierten Klassen wird aber am Beispiel deutlich: So wurde StockMash einerseits nur mit Links modelliert, andererseits unter Einsatz der BackLinks. Während zur Modellierung im einfachen Fall 19

Verbindungen nötig waren, konnte die Komplexität des Modells mit BackLinks bei gleicher Funktionalität auf nur acht Verbindungen reduziert werden.
Weiterhin konnte mit der Anwendung die Abbildung von Drag-and-Drop auf das Kommunikationsmodell gezeigt werden. Alle angestrebten und konzipierten Kommunikationsformen konnten somit erfolgreich umgesetzt werden.

7.3.3 Adaptive Touristeninformation mit *TravelGuide*

Mit der Anwendung *TravelGuide* in Abbildung 7.11 exemplarisch die Umsetzung der in Abschnitt 5.3 vorgestellten Adaptionstechniken und -aspekte gezeigt. Grundlage für die Realisierung im Rahmen einer studentischen Arbeit (vgl. Radeck (2010)) bot das Szenario aus Abschnitt 2.2.3. Die Anwendung dient der Erkundung einer Stadt und bietet dem Nutzer Informationen zur Sehenswürdigkeiten in seiner näheren Umgebung, wobei Kontextinformationen hinsichtlich seiner Person und seines Endgerätes berücksichtigt werden.

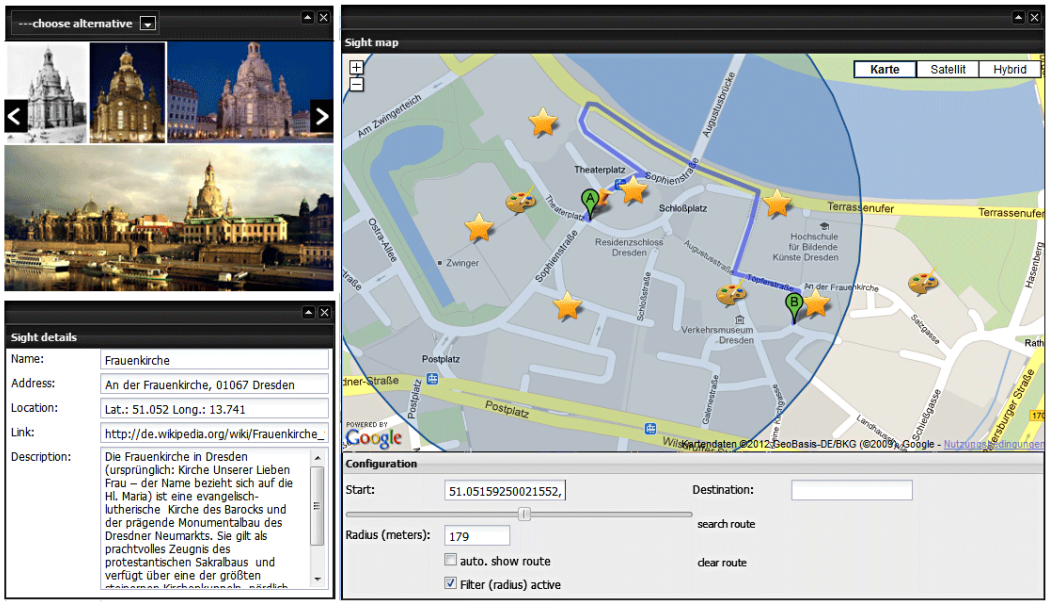

Abb. 7.11: Adaptiver Touristenführer *TravelGuide*

Zur Realisierung der gewünschten Funktionalität wurden eine Reihe von Komponenten entwickelt. So werden u. a. Wikipedia, der Bilderdienst von Google und YouTube angebunden, um Informationen zu den entsprechenden Ausflugsziehen zu beziehen. Diese werden durch UI-Komponenten adäquat visualisiert, wozu u. a. eine Google Map [@Google, 2011b], ein Video-Player, eine Bildergalerie sowie ein Formular für die Detailinformationen umgesetzt wurden.
Die Konfiguration, Verknüpfung und Anordnung der Komponenten erfolgt, wie bei den obigen Anwendungen, unter Nutzung der in Kapitel 5.2 vorgestellten Modellkonstrukte. Ein besonderes Augenmerk bei der folgenden Betrachtung gilt dem kontextadaptiven Verhalten, welches durch das Adaptionsmodell beschrieben und im Zusammenspiel mit der TSR umgesetzt wurde.

Abbildung 7.12 zeigt einige Ausschnitte des Kompositionsmodells von TravelGuide im grafischen Editor. Auf der linken Seite (A) sind u. a. die Komponenten der Anwendung

sowie im unteren Bereich die verschiedenen *ContextualScreenflows* zu erkennen, die hier auf Endgeräte mit verschiedener Displaygröße abgestimmt sind. Die rechte Seite (B) gibt einen Einblick in das Adaptivity Model. Dort ist zunächst ein Adaptionsaspekt *LowBattery* zu sehen, der den dynamischen Komponententausch am Beispiel der Komponente zur visuellen Präsentation von Sehenswürdigkeiten (links oben in Abbildung 7.11) nach sich zieht. Seine Modellierung wurde bereits in Abbildung 5.24 schematisch aufgezeigt: Ändert sich der Kontextparameter dev:BatteryLevel, wird durch die Schachtelung von *Logical-* und *ParameterConditions* geprüft, ob dieser unter 20% liegt und es sich um ein mobiles Endgerät handelt. Ist dies der Fall, erfolgt der Austausch der Videokomponente mit einer Bildergalerie. Der Zustandserhalt wird durch die ExchangeComponentAction, wie in Abschnitt 6.3.3.3 beschrieben, umgesetzt. Ist in der Anwendung beispielsweise ein Video des Dresdner Zwingers zu sehen, so wird dieses gegen eine Bildergalerie des Zwingers ausgetauscht.
Die Internationalisierung der Anwendung erfolgt durch den ContextLink *UserLanguage*, der darunter zu sehen ist. Er verbindet die semantisch als person:Language getypten Spracheigenschaften der drei UI-Komponenten von TravelMash mit dem entsprechenden Kontextparameter. Dieser ist weiter unten als Property *user:hasLanguage* von Context zu sehen. Da die Verbindung ein Attribut des ContextLinks darstellt, ist sie hier im Baum nicht explizit sichtbar. Der ContextLink ist als initial markiert, sodass bereits beim Laden der Anwendung die Sprache des Nutzers ausgelesen und zur Konfiguration aller Komponenten genutzt wird. Die Umsetzung der Sprachanpassung obliegt den Komponenten selbst, z. B. durch das Laden verschiedener Sprachpakete bei Änderungen ihrer Spracheigenschaft.
Auch die Anpassung des aktuellen Standorts des Nutzers auf der Karte wird über einen ContextLink (*UserLocation*) modelliert. Die entsprechende Eigenschaft $currentLocation$ wird wiederum mit der Nutzerposition $UserLocation$ der Kontextkomponente verbunden. Der Kontextlink gilt permanent, d. h. mit jeder Änderung im Kontextmodell erfolgt die Synchronisation mit der Eigenschaft der Karte und somit das Umsetzen des Markers. Die Reaktion, z. B. das Laden neuer Daten zu Sehenswürdigkeiten im Umfeld, obliegt der Kartenkomponente selbst.

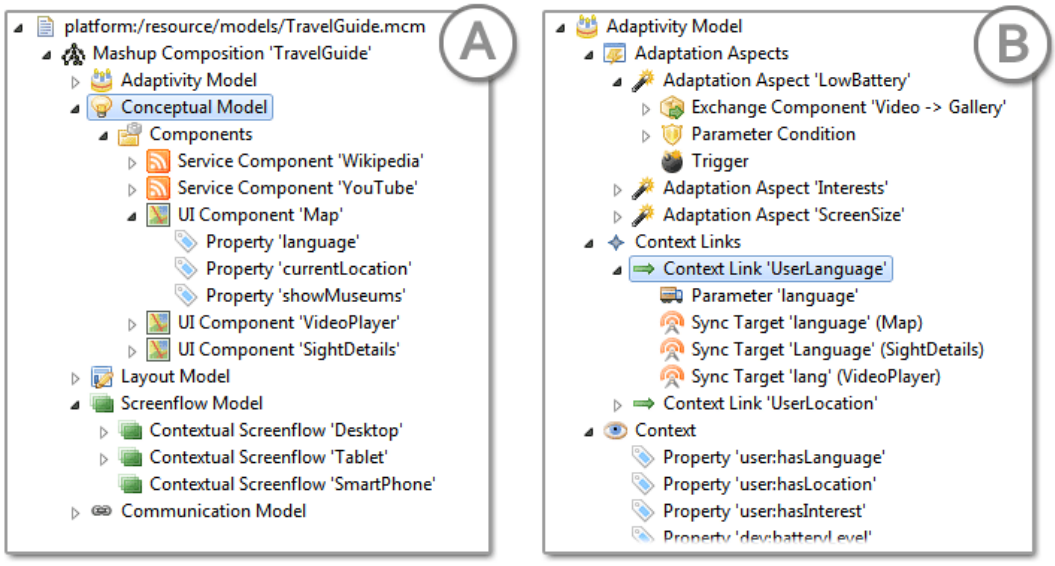

Abb. 7.12: Teile des *TravelGuide*-Modells im grafischen Editor

Andere Adaptionen, wie die Anzeige von Sehenswürdigkeiten im Abhängigkeit von Nutzerinteressen, wurden über Adaptionsaspekte modelliert. So sorgt der Aspekt *Interests* dafür, dass bei Änderungen der Nutzerinteressen – hier ist u. a. der Kontextparameter hasMuseumInterest relevant – die Kartenkomponente rekonfiguriert wird, sodass sie Museen ein- oder ausblendet. Die Anpassung an die verfügbare Darstellungsfläche ist im Aspekt *ScreenSize* modelliert. Dieser wird durch Veränderungen der Bildschirmbreite ausgelöst (Trigger). Über eine ParameterCondition wird geprüft, ob diese unter 600 Pixeln liegt. Ist dies der Fall, erfolgt die Rekonfiguration des Layouts von einer zwei- zu eine einspaltigen Version. Alternativ kann als Advice der Wechsel zu einem anderen ContextualScreenflow modelliert werden, der zur Auswahl eines auflösungsgerechten Views und somit ebenfalls zu einer Layout-Anpassung führen würde.

Mit der erfolgreichen Umsetzung der Anwendung und der Unterstützung aller im Szenario formulierten Anforderungen konnten die Adaptionskonzepte bezüglich der Modellierung und Umsetzung validiert werden. Es wurden alle in Abschnitt 5.3.1 identifizierten Adaptionstechniken prototypisch realisiert und getestet – einzig für die dynamische Adaption von Screenflows steht die Umsetzung noch aus. Die „Adaption der Komponentenschnittstellen“ wurde nicht explizit mit Modellkonstrukten bedacht, da durch die Laufzeitumgebungen bereits inhärent in Form der in Abschnitt 6.2.4.2 geschilderten Mediationsverfahren unterstützt werden. In diesem Zusammenhang ist außerdem anzumerken, dass bereits die Auswahl der Komponenten zur Initialisierungszeit kontextadaptiv, z. B. in Abhängigkeit von den Möglichkeiten des Endgerätes, wie verfügbaren Browser-Plug-ins, erfolgt.

7.3.4 Weitere Prototypen

Neben den beschriebenen Anwendungen wurden eine Reihe weiterer Prototypen umgesetzt, von denen zwei in Abbildung 7.13 dargestellt sind.

Abb. 7.13: Weitere Prototypen

Auf der linken Seite ist eine der ersten Beispielanwendungen zu sehen, die mit einer frühen Umsetzung der vorgestellten Konzepte realisiert wurde. Sie ermöglicht die Verwaltung von Kontakten und bietet zusätzliche Informationen über ihren aktuellen Aufenthaltsort. Nutzer können die Daten in einem Formular editieren und den Ort direkt auf der Karte verändern.
Auf der rechten Seite ist die bereits erwähnte **CRM-Anwendung** zu sehen, die dem Nachweis der serverseitigen Komposition auf Basis der RAP-basierten MRE diente. Für den Test der Kommunikation mit Rahmenanwendungen bzw. der

Plattform bezieht sie die darzustellenden Kundeninformationen von der umgebenden Anwendung auf Basis von CAS Open. Weiterhin konnte in diesem Zusammenhang die plattformübergreifende Nutzung von Komponenten validiert und getestet werden, da durch die angesprochenen Mechanismen zur automatischen Kapselung (vgl. Abschnitt 7.2.2.2) diverse clientseitige Komponente auch für die serverseitig komponierte Anwendung zum Einsatz kommen konnten.

Schließlich wurde auch die **Geschäftsprozessintegration** auf Basis der HT-MRE anhand eines beispielhaften Prozesses erprobt und validiert. Dieser wurde mit dem ActiveVOS Designer [@ActiveVOS, 2011] entworfen und spiegelt die Aufnahme eines Versicherungsfalls bei Gebäudeschäden wieder. Die Umsetzung erfolgte als BPEL-Prozess mit diversen menschlichen Aktivitäten, die ihrerseits als komposite Webanwendung auf Basis der vorgestellten Konzepte modelliert und in die Task-Beschreibungen integriert wurden. Genauere Informationen zu Szenario und Modellierung können einer separaten Veröffentlichung [Pietschmann et al., 2009a] entnommen werden.
Die Interaktion mit dem Prozess erfolgt in mehreren, aufeinanderfolgenden Schritten und durch verschiedene Rollen, z. B. die Begutachtung durch Außendienstmitarbeiter und die Evaluation des Falls durch „Entscheider“. Beide Parteien benötigen Informationen aus dem Prozess, die adäquat visualisiert werden müssen, z. B. Kundendaten, die Schadensbeschreibung (textuell und anhand von Fotografien), eine Route zur Anfahrt und Eingabemöglichkeiten für die Interaktion mit den Gutachter. Für jeden Interaktionsschritt wurde eine Komposition aus UI-Komponenten erstellt, die Zugriff auf die Daten des Prozesses bietet. Ihre Integration in die HumanTask-Beschreibung erfolgte über das *rendering*-Element in Form eines XSLT-Stylesheets, welches eine serialisierte Kompositionsbeschreibung zur Ausgabe hat.
Die erfolgreiche Integration und Ausführung in einer Geschäftsprozessumgebung zeigte, wie vielseitig die entwickelten Konzepte eingesetzt werden können, um den Entwicklungsprozess kompositer, interaktiver Anwendungen zu vereinfachen und zu beschleunigen. Statt der Programmierung dedizierter Task-Oberflächen konnten bestehende Komponenten verwendet und bedarfsgerecht komponiert werden.

Mit Hilfe dieser und der zuvor genannten Beispielanwendungen konnten die Validität und Praktikabilität des vorgestellten, modellgetriebenen Entwicklungsprozesses und der Laufzeitarchitektur für verschiedene Anwendungsszenarios und Plattformen nachgewiesen werden. Der folgende Abschnitt fasst die Beiträge der letzten Abschnitte zusammen und gibt genaueren Aufschluss über die Relation zu den identifizierten funktionalen und nicht-funktionalen Anforderungen an die Konzeption.

7.4 Zusammenfassung und Diskussion

Nachdem in Kapitel 5 und 6 die neuen wissenschaftlichen Konzepte zur modellgetriebenen Entwicklung adaptiver, kompositer Webanwendungen vorgestellt wurden, widmete sich dieses Kapitel ihrer Realisierung und Validierung.
Dazu wurde zunächst ein Überblick über die prototypische Umsetzung der Modellierungsmittel gegeben, die die plattformunabhängige Repräsentation universeller Komponenten und Kompositionen ermöglichen. Danach rückte die Kompositionsinfrastruktur in den Mittelpunkt der Betrachtung. Nachdem die grundlegenden

Infrastrukturdienste zur Verwaltung und dynamischen Suche von Komponenten vorgestellt wurden, standen die verschiedenen, realisierten Laufzeitumgebungen und die Implementierung des Adaptionssystems im Mittelpunkt. Im dritten Teil des Kapitels konnten schließlich – ausgehend von den Anwendungsszenarien aus Abschnitt 2.2 – anhand von Beispielanwendungen die Validität und Praktikabilität der Konzepte nachgewiesen werden.
Im Folgenden werden die Ergebnisse und Beiträge des Kapitels zusammengefasst. Die Berücksichtigung und Erfüllung der Anforderungen aus Abschnitt 2.3 wurde bereits im Rahmen der Diskussion der Konzepte in den letzten beiden Kapiteln erläutert. Die folgenden Abschnitte greifen deshalb nur die wichtigsten Anforderungen im Kontext der Implementierung auf, die jeweils mit ✪ markiert sind.

Umsetzung der Modellierungskonzepte

Teilkapitel 7.1 widmete sich der Realisierung der Modellierungskonzepte aus Kapitel 5. Zunächst wurde auf die Formalisierung des neuen, universellen Komponentenmodells aus Abschnitt 5.1 bzw. die in diesem Zusammenhang vorgestellten Beschreibungsarten eingegangen. Dazu kommen mit XML-Schema und OWL DL zwei standardisierte Modellierungssprachen zum Einsatz. Mit ihrer Hilfe kann die ✪ Beschreibung von Komponenten wie in Abschnitt 5.1.3 vorgesehen umgesetzt werden. Ihre Erweiterbarkeit, u. a. durch Namensräume, erwies sich als gute Ausgangsbasis bei der iterativen Entwicklung des Vokabulars. Anhand einer Vielzahl von Komponenten, die im Rahmen von Demonstratoren und für Tests entwickelt wurden, konnte dessen Einfachheit und Ausdrucksstärke nachgewiesen werden.
Auch hinsichtlich der Verarbeitung der Komponentenbeschreibungen konnte auf etablierte Vorgehensweisen aufgebaut werden, u. a. durch die Erstellung einer Modell-API über die Referenzimplementierung von JAXB und Jena. Somit steht eine zentrale Schnittstelle zur Verfügung, die die effiziente programmatische Be- und Verarbeitung von (S)MCDL in Java erlaubt.
Die Realisierung der Konzepte aus Abschnitt 5.2 und Abschnitt 5.3 erfolgte nach Abwägung verschiedener Designalternativen als dediziertes Metamodell auf Basis von Eclipse EMF. Die ✪ Standardkonformität zur MOF und die Serialisierung als XMI gewährleisten sowohl die ✪ Plattformunabhängigkeit als auch die ✪ Interoperabilität von Kompositionsmodellen im Sinne der werkzeugübergreifenden Nutzung. Gleiches gilt für die in OCL formulierten, inhärenten Modell-Constraints.
Schließlich bot die Verwendung von EMF weitere Vorteile, wie die Bereitstellung einer Java-API zur Be- und Verarbeitung des Modells. Auf dieser Basis konnten die in Abschnitt 5.4 angesprochenen Unterstützungsmechanismen für den Autorenprozess umgesetzt werden. Sie umfassen u. a. automatische Modelltransformationen auf Basis von QVTO – ebenfalls Teil der MOF-Spezifikation – sowie die Abbildung in ausführbare Webanwendungen mittels Transformationslogik in Xpand. Zur Vereinfachung von Entwicklung und Test wurde außerdem ein Autorenwerkzeug entwickelt, welches die grafische Modellierung kompositer Webanwendungen erlaubt.

Die in dieser Arbeit vorgestellten, neuen Konzepte zur plattformunabhängigen Modellierung universeller Kompositionen konnten somit vollständig umgesetzt werden. Die Repräsentation von verteilten Web-Ressourcen nach dem universellen Komponentenmodell und deren Komposition zu interaktiven Webanwendungen erwiesen sich als praktikabel und konnten anhand verschiedener Beispiele erprobt

werden. Mit der Formalisierung der Modelle konnte der Grundstein für die angestrebte, modellgetriebene Entwicklung kompositer, interaktiver Anwendungen gelegt werden. Die Realisierung dieser Vision für Endnutzer macht jedoch eine umfassendere Werkzeugunterstützung nötig, die nicht im Fokus dieser Arbeit stand.

Realisierung von Kompositionsinfrastruktur und Laufzeitumgebungen

In Abschnitt 7.2 wurde die Realisierung der verschiedenen Module der serviceorientierten Kompositionsinfrastruktur beschrieben.

Zur **Komponentenverwaltung** wurde das CoRe als Web Service realisiert, welches über SOAP plattformunabhängig angebunden werden kann und verschiedenen MREs und Autorenwerkzeugen zur Verfügung steht. Die ✪ Plattformunabhängigkeit wird zudem durch die Nutzung standardkonformer Abfragesprachen, wie SPARQL, unterstützt. Als interne Verwaltungsgrundlage dient die MCDO, da sie alle vorgestellten Beschreibungsformen subsumiert. Der Zugriff auf semantische Typen, d. h. Groundings, Liftings und Lowerings, im Rahmen der Mediation erfolgte ebenfalls über eine Web-Service-Schnittstelle.

Die **Discovery-Verfahren** wurden nach der Abwägung verschiedener Verteilungsalternativen im Hinblick auf die Praktikabilität und ✪ Performanz als Teilmodule des CoRe umgesetzt. Über dessen SOAP-Schnittstelle können nun auch Template-Anfragen gestellt werden. Die implementierte Logik deckt die in Abschnitt 6.1 geschilderten Algorithmen vollständig ab – lediglich einige Modellerweiterungen der MCDO, d. h. gewichtete Modellreferenzen, Vor- und Nachbedingungen, werden bei der Auswertung von Templates bislang nicht berücksichtigt.

Bei der Implementierung wurde auf eine größtmögliche ✪ SoC zwischen den Modulen und der Grundfunktionalität von CoRe geachtet. Auch die ✪ Erweiterbarkeit, z. B. um andere Matching- und Ranking-Strategien oder Anfragemöglichkeiten, wurde sichergestellt. Die Nutzung asynchroner Tasks ermöglicht die effiziente, parallele Verarbeitung im Sinne der ✪ Performanz. Anhand von Referenz- bzw. Testfällen konnte die korrekte Funktionsfähigkeit der Discovery-Algorithmen unter Beweis gestellt werden. Lasttests am CoRe zeigten u. a. die ✪ Stabilität der Implementierung und den Vorteil der asynchronen Verarbeitung im Multi-Threading-Betrieb auf. Allerdings wurde dabei auch die direkte zeitliche Abhängigkeit der Discovery von der Größe der Komponentenmenge deutlich, was die Notwendigkeit weiterer Optimierungen, z. B. in Form von Caching und Verteilungsmechanismen, unterstreicht.

Als **Laufzeitumgebungen** für die Kompositionen wurde die in Abschnitt 6.2.2 vorgestellte Referenzarchitektur auf Basis verschiedener Technologien realisiert, um die ✪ Plattformunabhängigkeit der Konzepte zu demonstrieren. Für alle Lösungen erwies sich die direkte Interpretation des Kompositionsmodells als praktikabelster Weg. Im Gegensatz zur MDA kann bei der modellgetriebenen Entwicklung somit auf die Transformation in plattformspezifische Zwischenmodelle verzichtet werden.

Den Schwerpunkt der Umsetzung bildete die TSR zur clientseitigen Komposition und Ausführung. Sie setzt die Referenzarchitektur vollständig auf Basis des JavaScript-Frameworks Ext Core um. Allein der Mediator wurde aus Gründen der ✪ Performanz in einen Web Service ausgelagert, der auch anderen MRE-Implementierungen zur Verfügung steht.

Die Leistung der clientseitigen Laufzeitumgebung ist naturbedingt stark vom Browser und den Möglichkeiten des Endgeräts abhängig. Anhand der Beispielanwendungen

konnte jedoch gezeigt werden, dass u. a. in Google Chrome und Mozilla Firefox die TSR im Zusammenspiel mit den externen Diensten stabil und ohne signifikante Beeinträchtigungen für den Nutzer funktioniert. Der Einfluss von Modellinterpretation und dynamischer Komposition auf die Ladezeit erwies sich in Relation zur eigentlichen Ladezeit der Komponenten als vertretbar gering.
Vor dem Hintergrund der Anforderungen geschäftlichen Anwendungen wurde eine weitere MRE-Implementierung zur serverseitigen Integration und Komposition auf Basis von Eclipse RAP in Java entwickelt und in die Plattform CAS Open eines Industriepartners integriert. Die technologische Grundlage bildete das OSGi-Framework, dessen Mechanismen für die Modularisierung, die ✪ späte Bindung und das ✪ Life-Cycle-Management von *Bundles* im Sinne der Konzeption ausgenutzt werden konnten. Über eine Erweiterung des CoRe gelang zudem die Kapselung clientseitiger Komponenten als OSGi-Bundles, was ihre plattformübergreifende Nutzung in der TSR und der RAP-Umgebung ermöglichte.
Mit der erfolgreichen Umsetzung der RAP-MRE konnte u. a. die ✪ Plattformunabhängigkeit der Anwendungsmodelle nachgewiesen werden. Die Modellierung unabhängig von der technologischen Umsetzung oder Verortung auf Client oder Server, wird möglich, da die Integration jeweils kompatibler Komponenten durch die Discovery-Mechanismen sichergestellt wird. Das setzt freilich die Verfügbarkeit entsprechender Komponenten für alle potentiellen Ausführungsplattformen voraus.
Die Wahl zwischen client- oder serverseitiger Laufzeitumgebung muss je nach Anwendungsfall getroffen werden. Zwar lässt sich durch die serverseitige Komposition und Verarbeitung, die durch die Mehrzahl der Referenzmodelle postuliert wird, ein höheres Maß an Qualität und Sicherheit realisieren, sie skalieren mit steigernder Nutzerzahlen aber naturgemäß schlechter als die rein clientseitige Lösungen. Zudem wirkt sich der Umweg über den Server bei der Kommunikation zwischen Komponenten negativ auf die ✪ Performanz aus.
Mit der HumanTask-MRE wurde ein weiteres Einsatzszenario des neuen Kompositionsansatzes aufgezeigt. Gegenüber automatisch generierten Formularen oder der manuellen Erstellung der Task-Oberflächen können damit schneller und kostengünstiger maßgeschneiderte Oberflächen zur Nutzerinteraktion innerhalb von Geschäftsprozessen angeboten werden. Dazu wurde der TLC von ActiveBPEL so erweitert, dass er Kompositionen als alternative *Renderings* einzelner Tasks interpretiert und bereitstellt. Eine besondere Herausforderung war hierbei die Realisierung der Schnittstelle zum Geschäftsprozess, für die eine dedizierte UI-Komponente geschaffen wurde. Aufgrund der ✪ Standardkonformität zu WS-HumanTask, kann die HT-MRE auch mit anderen Geschäftsprozess-Engines eingesetzt werden.

Mit der Umsetzung dieser drei exemplarischen Laufzeitumgebungen konnte die Validität des technischen Konzeptes der modellgetriebenen Entwicklung universeller Kompositionen nachgewiesen werden. Für alle Plattformen bildet das entwickelte Kompositionsmodell den Ausgangspunkt zur dynamischen Integration von Mashup-Komponenten, die neben Geschäftslogik und Daten auch Benutzerschnittstellen enthalten können. Zur Komponenten- und Typverwaltung, Discovery und Kontextverwaltung konnte auf die bereits vorgestellte, modulare Infrastruktur aus Web Services zurückgegriffen werden.

Schließlich wurden die in Abschnitt 6.3 vorstellten **Adaptionskonzepte** realisiert. Grundlage hierfür bot das System CroCo zur ✪Kontextverwaltung, dessen Entkopplung von anderen Modulen und MREs im Sinne der ✪SoC durch die Realisierung als Web Service erfolgte. Die semantische Verwaltung der Kontextwissensbasis geht einher mit der semantischen Typisierung der Kompositionen, d. h. die einheitliche Nutzung semantischer Konzepte erlaubt die ✪Kontextsensitivität sowohl im Discovery-Prozess durch das CoRe als auch zur Laufzeit. Auch aus diesem Grund kommen für den Datenaustausch mit RDF und SPARQL zwei Standards zum Einsatz.
Die Realisierung des Adaptionssystems erfolgte in Verbindung mit der TSR auf Basis von JavaScript. Beide sind über den *Adaptation Manager* lose gekoppelt, der direkt in die ereignisorientierte Kommunikation der Anwendung und Laufzeitumgebung eingebunden ist. Ein Plug-In-Konzept erlaubt die Integration beliebiger, lokaler Kontextmonitore zur Erfassung relevanter Kontextdaten. Diese werden im Sinne der ✪Performanz lokal verwaltet und nur bei Bedarf mit dem entfernten Kontextmodell von CroCo synchronisiert. Der Adapter dazwischen unterstreicht die Unabhängigkeit des Adaptionssystems vom Kontextdienst. Allerdings resultiert diese Generik in einem erhöhten Aufwand bei der Konvertierung von Anfragen und Antworten aus bzw. in die internen Repräsentationen.
Für die Umsetzung der Context Links konnte auf die bestehende Link-Infrastruktur zurückgegriffen werden. Adaptionsaspekte wurden hingegen auf ECA-Regeln abgebildet, um die möglichst effiziente Auswertung zur Laufzeit zu garantieren. Bei den Test- und Beispielanwendungen waren keine signifikanten Beeinträchtigungen der ✪Performanz durch die dynamische Regelauswertung spürbar. Die auszulösenden Adaptionsaktionen wurden einerseits als abstrakte Aktionen im Adaptionssystem und andererseits als deren Implementierungen in der TSR realisiert. Somit kann die Unabhängigkeit der Regelauswertung von spezifischen MREs im Sinne der ✪SoC gewährleistet werden. Zur Vorbeugung von Konflikten zwischen Adaptionsaspekten und -techniken sind diese priorisierbar. Die Realisierung des Systems umfasst zusätzlich Adaptionssperren, um die gegenseitige Beeinflussung von Anpassungen bei der nebenläufigen Ausführung zu vermeiden.

Bis auf die dynamische Adaption des Screenflows konnten alle Adaptionstechniken realisiert und an Beispielanwendungen getestet werden. Die vorgestellten Konzepte der Modellierung und Realisierung dynamischer Adaption universeller Kompositionen wurden somit praktisch untermauert. Dies schließt auch den dynamischen Komponententausch samt Mechanismen zum ✪Zustandstransfer ein. Die Anwendung der vielfältigen Adaptionstechniken über verschiedene Anwendungsebenen hinweg stellt gegenüber existierenden Systemen zur Adaption von Web-Service-Kompositionen oder Benutzerschnittstellen ein Novum dar.

Validierung anhand von Beispielanwendungen

Im dritten Teilkapitel wurde das Zusammenspiel der realisierten Konzepte anhand von Beispielanwendungen gemäß den Szenarien aus Abschnitt 2.2 verdeutlicht. Mit ihrer Hilfe konnten die grundlegenden Annahmen und Designentscheidungen der Konzeption sowie die Tragfähigkeit des Ansatzes unter Beweis gestellt werden.
Die dafür umgesetzten Komponenten untermauerten zunächst die ✪Universalität des Komponentenmodells. Bestehende APIs und Web Services konnten ebenso nach den vorgestellten Prinzipien repräsentiert werden, wie visuelle Komponenten

in Form interaktiver Karten, Bildergalerien, etc. Ihre semantische Auszeichnung verdeutlichte die zentrale Rolle der Domänenmodelle im Konzept: hier werden möglichst universelle und ausgereifte Konzeptionalisierungen für domänenspezifische Daten und Funktionalitäten benötigt. Im Sinne der ✪Wiederverwendbarkeit und ✪Plattformunabhängigkeit wird deshalb auf ein etabliertes und teilweise standardisiertes Vokabular zurückgegriffen.
Mit der **Modellierung** der kompositen Anwendungen konnte dann die ausreichende Mächtigkeit des Modells geprüft und belegt werden, da alle funktionalen Anforderungen durch Modellklassen abgedeckt wurden. Den Basisfall stellte die Anwendung *TravelMash* dar, die die grundlegenden Anforderungen an die universelle Modellierung und deren Interpretation, dynamischen Integration und Ausführung adressierte. Die Umsetzung von *StockMash* diente insbesondere der Evaluation der Konzepte zur erweiterten ✪Koordination. So wurde u. a. die Funktionsfähigkeit der bidirektionalen Kommunikationsbeziehungen zur Datenanfrage und der Synchronisation belegt. Beide Anwendungen veranschaulichen außerdem die ✪Wiederverwendbarkeit von Komponenten, z. B. mittels generischer Listen (TravelMash). Durch die adaptive Anwendung *TravelGuide* wurde schließlich verdeutlicht, wie Adaptionslogik samt der identifizierten Adaptionstechniken in Form eines unabhängigen Teilmodells einer Komposition beschrieben werden kann.

Die ✪Abstraktion im Kompositionsmodell versetzt nun erstmals Domänenexperten in die Lage, interaktive Mashup-Anwendungen ohne Programmierung selbst zu gestalten. Die Anwendungskomplexität „verschwindet“ freilich nicht – sie wird vielmehr hinter die Komponentenschnittstellen verschoben. Dementsprechend zeigte sich bei der praktischen Realisierung möglichst generischer, gut wiederverwendbarer und robuster Komponenten ein erhöhter Aufwand für deren Entwickler. Dieser Mehraufwand macht sich schließlich bei der Anwendungskomposition bezahlt, da sich der Entwicklungsprozess im Idealfall auf die Suche nach passenden Komponenten und deren Komposition verkürzt. Voraussetzung dafür bleibt das Vorhandensein alternativer Komponentenimplementierungen oder die Nutzung von Kapselungsmechanismen, wie sie für die RAP-MRE entwickelt wurden.

Die erfolgreiche **Interpretation und Ausführung** der Anwendungen auf Basis der geschilderten Kompositionsinfrastruktur untermauerte die korrekte Umsetzung und Funktionsweise der Laufzeitkonzepte. Durch alternative Komponenten wurde u. a. die kontextsensitive Discovery erprobt: In TravelMash wurden abhängig vom verfügbaren Bildschirmplatz verschiedene Routenanzeigen ausgewählt und in StockMash erfolgte die Auswahl von Aktienvisualisierungen nach der Verfügbarkeit des Flash-Plug-ins im Browser. TravelGuide veranschaulichte vorrangig die praktische Relevanz und Realisierung von ContextLinks, z. B. zur Internationalisierung, sowie von diversen Adaptionsbelangen zur Personalisierung und Endgeräteanpassung. Die vorgestellten Adaptionsbelange konnte dabei erfolgreich am Beispiel validiert werden.
Gleichsam dienten die Anwendungen der praktischen Erprobung des vorgestellten Lebenszyklus' von Komponenten sowie ihrer uni- und bidirektionalen Verknüpfung über Ereignisse. Am Beispiel von Drag-and-Drop wurde deutlich, wie auch die Abbildung endgerätespezifischer Interaktionsformen auf das ereignisorientierte Modell erfolgen kann.

Messreihen am Beispiel von TravelMash untermauerten die gute ✪Performanz der clientseitigen MRE-Implementierung. Sie zeigten, dass der zeitliche Anteil der Modellinterpretation, dynamischen Suche und Integration von Komponenten in Relation zu deren Ladezeit vertretbar gering ist. Durch Optimierungsverfahren, z. B. parallele Anfragen an das CoRe, kann er weiter verkürzt werden. Einige weitere Prototypen rundeten die praktische Erprobung und Validierung der realisierten Konzepte auch für die anderen MREs ab.

Zusammenfassend kann festgestellt werden, dass die neuen Konzepte zur universellen Komposition von Webanwendungen sowohl hinsichtlich der modellgetriebenen Entwicklung als auch hinsichtlich der dynamischen Bindung, Komposition, Ausführung und Adaption zu großen Teilen umgesetzt und anhand einer Vielzahl von Anwendungsbeispielen validiert werden konnten. Sie erwiesen sich als tragfähig und bieten gleichzeitig an diversen Stellen das Potential für Erweiterungen. Die Erfüllung der bereits in die Konzeption eingeflossenen Anforderungen aus Abschnitt 2.3 konnte im Rahmen der praktischen Realisierung und durch Tests bestätigt werden. Dabei wurden auch die nicht-funktionalen Anforderungen bedacht, z. B. hinsichtlich der ✪SoC, der einfachen ✪Erweiterbarkeit der geschaffenen Lösungen und der umfangreichen ✪Standardkonformität.
Das nächste Kapitel fasst die wissenschaftlichen Beiträge der Arbeit zusammen und skizziert ihr Potential als Ausgangspunkt für ausgewählte, weiterführende Arbeiten.

8

Zusammenfassung, Diskussion und Ausblick

In der vorliegenden Arbeit wurde ein umfassendes Konzept zur modellgetriebenen Entwicklung kompositer Web- bzw. Mashup-Anwendungen vorgestellt. Zunächst wurden systematisch die Probleme aktueller Entwicklungsansätze identifiziert und dementsprechend Arbeitsthesen und Forschungsziele zu ihrer Bewältigung formuliert. Ausgehend von verschiedenen Anwendungsszenarien wurden konkrete Anforderungen an das Lösungskonzept formuliert. Diese dienten gleichzeitig als Bewertungskriterien für die eingehende Untersuchung des Standes der Forschung und Technik, die aus zwei komplementären Perspektiven heraus vorgenommen wurde: Mit der Analyse von Lösungsansätzen für serviceorientierte Systeme auf der einen und interaktiven Webanwendungen auf der anderen Seite, konnten die Probleme und Defizite bestehender Konzepte deutlich gemacht werden.
Ausgehend von den identifizierten Anforderungen, Defiziten und einigen vielversprechenden Konzepten aus verwandten Arbeiten wurde in Kapitel 4 zunächst das Gesamtkonzept der modellgetriebenen Entwicklung interaktiver, kompositer Webanwendungen in seinen Grundzügen vorgestellt. Die Betrachtung im Detail erfolgte im Anschluss: Kapitel 5 widmete sich der Erläuterung der Modellierungskonzepte zur universellen und plattformunabhängigen Repräsentation von Komponenten und deren Kompositionen zu reichhaltigen, interaktiven Webanwendungen. Danach wurden in Kapitel 6 Konzepte für die dynamische, kontextabhängige Suche und Integration von Komponenten erarbeitet und eine Referenzarchitektur für ihre Ausführung vorgestellt. Die Kontextsensitivität bildete einen querschneidenden Belang, der konzeptionell in beide Kapitel einfloss.
Zuletzt bot Kapitel 7 einen Einblick in die praktische Umsetzung der Konzepte und zeigte anhand von Demonstratoren die Validität und Praktikabilität der geschaffenen Lösungen am Anwendungsbeispiel.

Die folgenden Abschnitte dienen der Diskussion der in dieser Arbeit erreichten, wissenschaftlichen Ergebnisse. Dazu werden in Abschnitt 8.1 die Inhalte und Beiträge der einzelnen Kapitel zusammengefasst. In Abschnitt 8.2 werden danach die Kernbeiträge der Arbeit in ihrer Gesamtheit diskutiert und hinsichtlich der in Kapitel 1 beschriebenen Problemstellungen, Thesen und Forschungsziele eingeordnet. Abschnitt 8.3 gibt abschließend einen Ausblick auf aktuelle und weiterführende Arbeiten, denen die Dissertation als Ausgangspunkt dienen kann.

8.1 Zusammenfassung der Kapitel und ihrer Beiträge

Die folgenden Abschnitte fassen die Kapitel der vorliegenden Arbeit in aller Kürze zusammen und verdeutlichen die jeweiligen Forschungsbeiträge.

Kapitel 1 | Einleitung

In Kapitel 1 wurden die Ziele dieser Arbeit herausgearbeitet und diverse Forschungsfragen aufgeworfen, die mit der Modellierung und Ausführung adaptiver, kompositer Webanwendungen einhergehen. Es wurde deutlich, dass sich mit dem Wandel des Internets zu einer Anwendungsplattform die Art und Weise, wie entsprechende Anwendungen entwickelt werden, ändern muss. Hier sieht sich eine leicht wachsende Anzahl von Programmierern dem drastisch steigenden Bedarf an Anwendungen für verschiedenste Endgeräte- und Nutzerklassen gegenüber. Als ein potentieller Ausweg wurde das Paradigma der Mashups zur universellen Komposition bedarfsgerechter Anwendungen aus bestehenden Komponenten angeführt. Zusammen mit Abstraktionsmechanismen kann es dazu beitragen, die Entwicklung zu vereinfachen, zu beschleunigen, und langfristig auf Nicht-Programmierer bzw. Fachexperten auszulagern. Zur Erfüllung dieser Vision wurden diverse, bislang ungelöste Forschungsprobleme identifiziert. Davon ausgehend wurden die Thesen und Forschungsziele dieser Arbeit formuliert und ein Überblick über die restlichen Kapitel gegeben.

Kapitel 2 | Grundlagen, Szenarien und Herausforderungen

Das zweite Kapitel diente der begrifflichen und fachlichen Einordnung der Arbeit in den Forschungskontext. Dazu wurden zunächst zentrale Begrifflichkeiten sowie konzeptionelle Grundlagen der Arbeit geklärt. Zur Veranschaulichung der konzeptionellen und praktischen Herausforderungen wurde exemplarisch drei Anwendungsszenarien vorgestellt und diskutiert. Sie bildeten auch den Ausgangspunkt für die spätere, praktische Validierung. Auf der Grundlage dieser Analyse wurden konkrete Anforderungen an die zu entwickelnden Design- und Laufzeitkonzepte formuliert, die gleichzeitig der Bewertung des aktuellen Standes der Forschung und Technik im nächsten Kapitel dienten.

Forschungsbeiträge: Das Kapitel bietet einerseits eine kompakte Einordnung der angestrebten Ziele in den Forschungskontext. Die Ableitung konkreter Anforderungen an die universelle Modellierung, Komposition und Ausführung adaptiver, kompositer Webanwendungen bildet einen weiteren wissenschaftlichen Beitrag.

Kapitel 3 | Stand der Forschung und Technik

Ziel von Kapitel 3 war es, dem Leser einen Überblick über den Stand der Forschung und Technik auf den betroffenen Forschungsgebieten zu geben. Die Analyse der relevanten Arbeiten erfolgte aus zwei Blickwinkeln: Zum einen wurden Ansätze aus dem Umfeld serviceorientierter Architekturen (SOA) untersucht, die bereits vielversprechende Konzepte zur Anwendungskomposition, späten Bindung und losen Kopplung für die Daten- und Geschäftslogikebene bieten. Zum anderen wurden Web-Engineering-Ansätze betrachtet, die im Hinblick auf die Interaktivität der resultierenden Anwendungen komplementäre Lösungen bieten und mit leichtgewichtigen Mashup-Modellen einen bewussten Gegenpol zu den Kompositionsansätzen der SOA darstellen. Die Betrachtung macht die Defizite beider Forschungsrichtungen deutlich

und unterstreicht die zuvor geschilderten Herausforderungen. Einige Ansätze bieten zwar vielversprechende Teillösungen, wie die plattformunabhängige Kompositionsbeschreibung oder die dynamische Bindung ihrer Bestandteile, allerdings existiert keine ganzheitliche Lösung im Sinne der Arbeit. Die gewonnenen Erkenntnisse dienten als Grundlage für die Konzeption in den folgenden Kapiteln.

Forschungsbeiträge: Ausgehend von einer ausführlichen Recherche liefert das Kapitel eine zusammenfassende Darstellung und Bewertung aktueller Entwicklungs- und Kompositionsansätze aus den Bereichen SOA und Web-Engineering. Die Diskussion verschiedenster Lösungen und ihr Abgleich mit den zuvor formulierten Anforderungen zeigt die vorhandenen Defizite bei der Erstellung interaktiver Webanwendungen auf.

Kapitel 4 | Universelle Komposition adaptiver Webanwendungen

Als Lösungsansatz wurde in Kapitel 4 das Gesamtkonzept der modellgetriebenen Entwicklung adaptiver, kompositer Webanwendungen geschildert. Ausgangspunkt hierfür bildete die Darstellung eines neuen Rollenmodells, welches erstmals die Entwicklung, Bereitstellung und spätere Integration von UI-Bestandteilen als gleichwertige Anwendungsbausteine nach dem Dienstprinzip vorsieht. In der Folge wurde ein Überblick über die innovativen Kernkonzepte der Arbeit gegeben, die sich an den zu Beginn der Arbeit formulierten Herausforderungen orientieren:

❶ Ein plattformunabhängiges Metamodell, welches die belangorientierte Spezifikation interaktiver, kompositer Webanwendungen auf Basis eines universellen Komponentenmodells ermöglicht.

❷ Eine modulare, serviceorientierte Kompositionsinfrastruktur und Referenzarchitektur zur Interpretation des o. g. Modells, zur dynamischen Suche, Auswahl und Bindung von Komponenten sowie zur Ausführung der Anwendungen.

❸ Modellierungs- und Laufzeitkonzepte zur Unterstützung der Kontextadaptivität von Webanwendungen auf Komponenten- und Kompositionsebene.

Den ersten beiden Schwerpunkten wurde im Folgenden jeweils ein eigenes Kapitel gewidmet. Die Adaptionskonzepte bilden einen querschneidenden Belang, der sowohl im Rahmen der Modellierung als auch als Teil der Infrastruktur vorgestellt wird.

Forschungsbeiträge: Das Kapitel stellt ein neues Gesamtkonzept für komposite, adaptive Webanwendungen vor, welches erstmals die strukturierte Entwicklung universeller Kompositionen auf Basis eines plattformunabhängigen Modells, deren Ausführung und dynamische Anpassung erlaubt. Es veranschaulicht insbesondere das gegenüber dem traditionellen SOA-Verständnis erweiterte Rollenmodell und die Zusammenhänge zwischen den Design- und Laufzeitkonzepten, die in den folgenden Kapiteln jeweils im Mittelpunkt stehen.
Das Gesamtkonzept wurde im Rahmen einer Journalveröffentlichung [Pietschmann, 2009] vorgestellt, der internationale Publikationen zum universellen Kompositionsprinzip in [Pietschmann et al., 2009b] und [Pietschmann et al., 2009c] vorausgingen.

Kapitel 5 | Belangorientierte Modellierung adaptiver, kompositer Webanwendungen

Kapitel 5 widmete sich dem Entwicklungsprozess adaptiver, komponentenbasierter Mashup-Anwendungen mit dem Schwerpunkt der Modellierung. Grundlage dafür

bildete ein zustandsbasiertes Komponentenmodell, welches die einheitliche Repräsentation und Beschreibung verteilter Anwendungsbausteine der Daten-, Geschäftslogik- und Präsentationsebene erlaubt. Auf der Basis dieser Abstraktionen wurde ein belangorientiertes Metamodell konzipiert, das die Verknüpfung der Bausteine zu einer kompositen Anwendung beschreibt. Die Modellierung erfolgt plattformunabhängig und definiert entsprechend abstrakt die funktionalen Zusammenhänge, d. h. die anwendungsspezifische Konfiguration der Komponenten, den Daten- und Kontrollfluss, und das Layout der Oberfläche. Mit dem Adaptionsmodell wurde ein unabhängiges Teilmodell geschaffen, welches die abstrakte Spezifikation von Adaptionslogik in Form von Aspekten ermöglicht. So können vielfältige, kontextadaptive Konfigurationen und Anpassungen von Kompositionen beschrieben werden, die zur Laufzeit ausgeführt werden sollen.

Forschungsbeiträge: Das Kapitel führt zunächst ein neues, universelles Komponentenmodell ein, welches es erlaubt, verteilte funktionale Bestandteile verschiedener Anwendungsebenen einheitlich zu repräsentieren und abstrakt zu beschreiben. Einen substanziellen wissenschaftlichen Beitrag bildet die Spezifikation des belangorientierten Kompositionsmodells, das die strukturierte, plattformunabhängige Repräsentation kompositer, interaktiver Webanwendungen erlaubt. Es beschreibt die bedarfsgerechte, lose Kopplung von Komponenten im Front- und Backend zu individuellen Anwendungen. Mit dem Adaptionsmodell wurde schließlich erstmals ein Vokabular zur Spezifikation von kontextadaptivem Verhalten für derartige Kompositionen entwickelt.

Alle Beiträge wurden auf einschlägigen, internationalen Tagungen vorgestellt, u. a. der Entwicklungsprozess in [Pietschmann, 2009] und das Metamodell in [Pietschmann et al., 2010a].

Kapitel 6 | Kontextsensitiver Integrationsprozess und Kompositionsinfrastruktur

Gegenstand von Kapitel 6 war die Konzeption einer verteilten Kompositionsarchitektur zur Unterstützung des modellbasierten Entwicklungsprozesses zur Laufzeit. Dazu wurde eine modulare, serviceorientierte Infrastruktur entwickelt, die wissenschaftliche Konzepte zur einheitlichen, semantischen Verwaltung von Komponenten, zu ihrer dynamischen, kontextabhängigen Suche, Auswahl und Bindung sowie zur Ausführung und dynamischen Adaption von universellen Kompositionen umfasst. Den ersten wesentlichen Bestandteil bildeten die Discovery-Algorithmen, welche die semantische Suche und Auswahl von Komponenten auf Basis der Abstraktionen im o. g. Modell realisieren. Darüber hinaus wurde die Referenzarchitektur einer MREs zur dynamischen Bindung und koordinierten Ausführung universeller Kompositionen vorgestellt. Die konzipierten Adaptionsmechanismen zeigten schließlich, wie die kontextsensitive Anpassung von Kompositionen zur Laufzeit auf Basis der abstrakten, aspektorientierten Beschreibung im Modell realisiert werden kann. Dabei wurde auch auf Konzepte zur Modellierung und Bereitstellung des nötigen Kontextwissens als Teil der Infrastruktur eingegangen.

Forschungsbeiträge: Ein Kernbeitrag des Kapitels bildet die neue, modulare Infrastruktur zur universellen Komposition interaktiver Anwendungen auf Basis des abstrakten Anwendungsmodells. Das Teilkonzept der dynamischen Bindung von Komponenten – entlehnt aus dem SOA-Umfeld – findet dabei erstmals auch auf der Präsentationsebene Anwendung. In Verbindung mit den Discovery-Mechanismen

wird so die kontextsensitive Suche, Auswahl und Bindung für beliebige Anwendungskomponenten möglich. Ein weiterer, substanzieller Beitrag besteht in der Referenzarchitektur zur Ausführung der Kompositionen, die – auf unterschiedliche Plattformen angewandt – die dynamische Integration und lose Kopplung von Komponenten erlaubt. Mit dem vorgestellten, unabhängigen Adaptionssystem wird der MRE ein Mechanismus zur Seite gestellt, der mit Hilfe innovativer Adaptionstechniken erstmals die Kontextualisierung universeller Komposition zur Laufzeit ermöglicht.
Auch diese Konzepte konnten im Rahmen internationaler Konferenzen und Workshops präsentiert werden. Die Veröffentlichungen umfassen u. a. den semantikbasierten Integrationsprozess [Pietschmann et al., 2011b], die MRE mit Fokus auf die clientseitige Komposition [Pietschmann et al., 2010b] und die Integration mit Geschäftsprozessen [Pietschmann et al., 2009a]. Die identifizierten Adaptionstechniken und die entsprechenden Konzepte wurden in [Pietschmann et al., 2011a] publiziert, und die zugrunde liegenden Mechanismen zur Kontextverwaltung mit CroCo in [Pietschmann et al., 2008] und [Mitschick et al., 2010] erläutert.

Kapitel 7 | Umsetzung und Validierung der Konzepte

Das letzte Kapitel gab einen Überblick über die technische Realisierung der entwickelten Konzepte. So wurde zunächst die Formalisierung von Komponenten- und Kompositionsmodell samt entsprechender Unterstützungsmechanismen für den Autorenprozess vorgestellt. Weiterhin wurde auf die prototypische Umsetzung der serviceorientierten Kompositionsinfrastruktur eingegangen, die unabhängige Dienste zur Verwaltung und semantischen Suche von Komponenten einschließt. Der Nachweis der Plattformunabhängigkeit konnte durch die Erprobung der Konzepte anhand verschiedener Kompositionsplattformen bzw. MREs erbracht werden. Schließlich wurde die Implementierung der konzipierten Adaptionsmechanismen erläutert, die in Verbindung mit der clientseitigen Laufzeitumgebung zum Einsatz kamen. Performanzanalysen untermauerten an verschiedenen Stellen die Wirksamkeit und Leistungsfähigkeit der umgesetzten Konzepte. Die Erprobung anhand von drei Beispielanwendungen mit Bezug auf die o. g. Referenzszenarien veranschaulichte schließlich, dass diese den identifizierten Herausforderungen gerecht werden.

Forschungsbeiträge: Mit der Realisierung von Modellen, Kompositionsinfrastruktur und diversen Demonstratoren konnten die Umsetzbarkeit, Validität und Praktikabilität der vorgestellten Konzepte im Hinblick auf die zu Beginn der Arbeit formulierten Ziele und Anforderungen belegt werden. Bis auf wenige Ausnahmen, die den Rahmen der Arbeit gesprengt hätten, konnten alle vorgestellten Konzepte implementiert und an Beispielanwendungen praktisch erprobt werden. Aufgrund der konsequenten Beachtung der nicht-funktionalen Anforderungen, u. a. hinsichtlich der SoC, Erweiterbarkeit und Standardkonformität, bilden die geschaffenen Lösungen eine solide Grundlage für die weitere Evaluation der Ergebnisse sowie für Erweiterungen im Rahmen angeschlossener Forschungsprojekte.
Auch die Ergebnisse der praktischen Umsetzung und Validierung wurden an verschiedener Stelle publiziert. So wurden u. a. Details zur Formalisierung des Kompositionsmodells [Pietschmann et al., 2010a], zur serverseitigen [Pietschmann et al., 2009b] und clientseitigen Kompositionsplattform [Pietschmann et al., 2010b] sowie zur Integration mit Geschäftsprozessen [Pietschmann et al., 2009a] veröffentlicht.

8.2 Diskussion und Bewertung

Im letzten Abschnitt wurden die Inhalte und Forschungsbeiträge der einzelnen Kapitel dieser Arbeit zusammengefasst. Alle Kernkonzepte wurden zudem, wie an entsprechender Stelle vermerkt, im Rahmen internationaler Veröffentlichungen publiziert. Eine ausführliche Auflistung kann dem Verzeichnis eigener Publikationen entnommen werden. Die folgenden Abschnitte fassen die wesentlichen, wissenschaftlichen Beiträge und Grenzen der geschaffenen Lösungen im Hinblick auf die Thesen und Ziele der Arbeit zusammen.

8.2.1 Wissenschaftliche Beiträge

Als Leitfaden für die Diskussion und Bewertung der Beiträge müssen die zu Beginn der Arbeit formulierten Probleme und Thesen herangezogen werden, die logisch aufeinander aufbauen und in den zentralen Forschungszielen aus Abschnitt 1.1.3 münden. Bereits im letzten Abschnitt wurde deutlich, dass diese Ziele vollständig erreicht wurden. Die dargestellten, wissenschaftlichen Ergebnisse bzw. Beiträge werden im Folgenden diskutiert und zueinander sowie zu den drei zentralen Herausforderungen der Arbeit in Beziehung gesetzt. Dabei wird insbesondere auf die Adressierung der identifizierten Probleme eingegangen, die als Beleg für die Arbeitsthesen dieser Dissertation dienen.

Herausforderung ❶: Modellierung

Universelles Komponentenmodell

Das Ziel der universellen Repräsentation aller Bestandteile kompositer Webanwendungen wurde erreicht und die entsprechenden Konzepte ausführlich in Abschnitt 5.1 beschrieben. Sie ermöglichen es erstmals, Web-Ressourcen und -Dienste aller Anwendungsebenen auf Basis der gleichen Abstraktionen zu repräsentieren, was ihre einheitliche, bedarfsgerechte Verknüpfung zu interaktiven, kompositen Webanwendungen erlaubt und die Vorteile aus These 1 nutzbar macht.
Im Gegensatz zur Mehrzahl bestehender Ansätze geht das Modell von zustandsbehafteten Komponenten aus und wird so insbesondere den Anforderungen interaktiver Bestandteile gerecht. Diese können nun ebenfalls – wie von Web Services bekannt – unabhängig von Anwendungen entwickelt und gewartet werden.
Die konsequente Durchsetzung des mit der Komponentisierung verbundenen *Black-Box*-Prinzips geht einher mit der technologieunabhängigen Beschreibung ihrer Schnittstellen. Über diese können Komponenten in verschiedensten Anwendungskontexten und Plattformen wiederverwendet bzw. „gebunden“ werden, was zur Senkung von Aufwand und Kosten bei der Anwendungsentwicklung beiträgt.
Über die Abstraktionen des Komponentenmodells wird gleichsam die technologische Komplexität verringert: Die An- und Einbindung von Komponenten bleibt nicht mehr nur Programmierern vorbehalten, sondern kann – entsprechende Autorenwerkzeuge vorausgesetzt – auch durch Domänenexperten erfolgen.
Mit der semantischen Auszeichnung von Komponenten wurde außerdem der Grundstein für intelligente Suchmechanismen und die gesteigerte Interoperabilität gelegt. So wird einerseits eine größere Treffermenge bei der Suche erreicht, andererseits können syntaktische Differenzen zur Laufzeit automatisch aufgelöst werden.

Plattformunabhängiges Kompositionsmodell

Auf Basis des in Abschnitt 5.2 vorgestellten, neuen Kompositionsmodells konnte die plattformunabhängige Repräsentation interaktiver Mashup-Anwendungen anhand diverser Beispielanwendungen anschaulich unter Beweis gestellt werden. Die Ergebnisse stützen die in These 2 formulierten Vorteile.
Ein Alleinstellungsmerkmal dieses innovativen Kompositionsansatzes besteht in seiner Universalität. Die einheitliche Repräsentation von Komponenten im Anwendungsmodell ermöglicht die bedarfsgerechte Komposition über alle Anwendungsebenen hinweg und bietet somit eine höhere Flexibilität als vorherrschende Lösungen: Einerseits können – je nach Bedarf – verschiedene UI-Komponenten bzw. Visualisierungen für die gleichen Hintergrunddienste gewählt werden. Andererseits wird der transparente Tausch von Datenquellen und Geschäftslogik „hinter" den Benutzerschnittstellen möglich.
Die Ausdrucksmächtigkeit des Modells geht über bisherige Ansätze aus den Bereichen SOA und Web Engineering hinaus. Neben der Definition und Konfiguration aller enthaltenen Komponenten und der Spezifikation des Daten- und Kontrollflusses umfasst es insbesondere Präsentationsaspekte, wie die visuelle Erscheinung und das Layout der Anwendung. Das Modell stützt sich dabei auf die Abstraktionen des o. g. Komponentenmodells, was mit diversen Vorteilen einhergeht:
Zum einen ist das abstrakt modellierte Kompositionswissen gegenüber programmatischen Lösungen besser wartbar und wiederverwendbar. Anwendungen werden über Plattform- und Endgerätegrenzen hinweg portabel, da sie durch verschiedene Laufzeitumgebungen interpretiert und ausgeführt werden können. Zum anderen kann die Auswahl konkreter Komponentenimplementierungen auf die Laufzeit verschoben werden, was die Einbeziehung von Kontextinformationen in den Kompositionsprozess ermöglicht. Schließlich versetzen die Abstraktionen erstmals Nicht-Programmierer bzw. Domänenexperten in die Lage, Anwendungslösungen zu entwickeln, da dabei vollständig auf die Programmierung verzichtet werden kann.
Diese Kombination aus der Flexibilität der Komposition und der nicht-programmatischen Entwicklung durch Domänenexperten erlaubt die Bedienung des *Long Tail* [Anderson, 2006] und stellt einen Lösungsansatz für die eingangs angesprochene *Impossible Equation* [Bezivin, 2011] dar. Die einfache, bedarfsgerechte Komposition situativer Softwarelösungen konnte an verschiedenen Beispielen prototypisch nachgewiesen werden. Im Idealfall kann der Entwicklungsaufwand auf die Suche, Konfiguration und Verknüpfung existierender Komponenten verkürzt werden.

Herausforderung ❷: Ausführung

Dynamische, kontextadaptive Bindung

Der in Abschnitt 6.1 vorgestellte Integrationsprozess ermöglicht erstmals die dynamische Suche und Integration von Komponenten beliebiger Anwendungsschichten in interaktive Webanwendungen und unterstreicht die mit These 3 verbundenen Vorteile. Das Prinzip der späten Bindung von Web Services konnte erfolgreich auf beliebige Anwendungsbestandteile, einschließlich Komponenten der Benutzerschnittstelle, angewendet und ein weiterer Beitrag zu deren Kontextualisierung geleistet werden. Die Erstellung kontextadaptiver Anwendungen – insbesondere Oberflächen – wird dadurch stark vereinfacht: Statt der Umsetzung verschiedener Varianten kann die

Funktionalität abstrakt modelliert und zur Laufzeit im Rahmen der Discovery auf die am besten passenden, kompatiblen Komponenten abgebildet werden.

Nach dem Vorbild der SWS bildet eine semantische Beschreibung den Ausgangspunkt für diese logikbasierte Suche. Durch die Berücksichtigung semantischer Teiltreffer bei der Suche wird eine gewisse Unschärfe hinsichtlich syntaktischer Eigenheiten von Komponenten möglich. Dies resultiert u. a. in einer größeren Treffermenge und potentiell besseren Komponenten. Zudem wird deren Interoperabilität gesteigert, was die Nutzung in verschiedenen Anwendungskontexten erlaubt und damit zur Senkung von Entwicklungskosten beiträgt. Die Schnittstellen und Daten gemäß dem definierten Anwendungsmodell werden automatisch auf die verfügbaren Komponenten abgebildet. Bestehende Ansätze zur semantischen Suche und Komposition in Mashups bieten dafür bislang keine Konzepte.

Neben dem Abgleich funktionaler Eigenschaften wird die Gewichtung der Kandidaten auch hinsichtlich nicht-funktionaler bzw. kontextspezifischer Kriterien unterstützt. Sie kann sowohl durch implizite Aspekte, z. B. die Kompatibilität der Kandidaten mit der Ausführungsumgebung, als auch durch den Kompositeur in Form kontextabhängiger Gewichtungsregeln beeinflusst werden. Dieser Auswahlprozess ermöglicht die Kontextadaptivität ausgelieferter Lösungen auf allen Anwendungsebenen.

Die dynamische Bindung hat einen weiteren Vorteil gegenüber statischen Ansätzen: Komponentenfehler können zeitnah und entkoppelt von Anwendungen behoben werden, sodass darin stets die aktuellsten und stabilsten Versionen von Komponenten zum Einsatz kommen. Diese ständige, unabhängige Wartung und Weiterentwicklung im Hintergrund wirkt sich – entsprechende Qualitätssicherung vorausgesetzt – positiv auf die Qualität der geschaffenen Anwendungslösungen aus.

Referenzarchitektur zur universellen Komposition

In Abschnitt 6.2 wurden sowohl eine modulare, serviceorientierte Kompositionsinfrastruktur als auch eine Referenzarchitektur zur Ausführung universeller Kompositionen vorgestellt. Sie runden das Konzept der modellgetriebenen Entwicklung kompositer Webanwendungen ab und verdeutlichen, dass deren Erstellung unabhängig von der späteren Zielplattform und ihrer Verortung zwischen Client- und Server erfolgen kann. Vielmehr werden Anwendungsmodelle durch verschiedene Implementierungen der vorgestellten Referenzarchitektur MRE interpretiert und ihre Abstraktionen somit auf die jeweiligen technologischen Plattformen abgebildet. These 2 und These 3 werden somit anschaulich untermauert.

Kompositionen bleiben ferner nicht länger auf eigens erstellte, lokale und bereits bekannte Komponenten beschränkt: Über die von allen MREs gemeinsam genutzte Infrastruktur in Form dedizierter Web Services kann auf ein potentiell größeres Repertoire verfügbarer Komponenten zurückgegriffen werden. Die dynamisch komponierten Anwendungen gewinnen dadurch an Reife und Qualität.

Aufgrund der oben geschilderten, verteilten Bereitstellung von Komponenten in Dienstform können diese durch die entwickelte Referenzarchitektur dynamisch gebunden bzw. integriert werden. Während dieser Mechanismus im Bereich der Daten und Geschäftslogik in Form von Web Services bereits Bestand hatte, stellt er für interaktive, zustandsbasierte Komponenten ein Novum dar. Im Gegensatz zu den weit verbreiteten generativen Ansätzen ist die Mächtigkeit der entstehenden Oberflächen somit nicht länger durch die Generierungslogik der Plattform beschränkt.

Die lose Kopplung der Komponenten bleibt indes auch zur Laufzeit gewahrt. Trotzdem bietet das technische Konzept der MRE gegenüber bisherigen Kompositionsplattformen deutlich mächtigere Koordinationsmuster, die insbesondere die Anforderungen interaktiver Anwendungen adressieren: So wird nicht nur die unidirektionale Verknüpfung von Ein- und Ausgängen unterstützt, sondern auch Request-Response-Beziehungen, permanente, asynchrone Aktualisierungen sowie die Synchronisation von Komponenten untereinander. Die konzeptionelle und praktische Abbildung von Drag-and-Drop auf das ereignisorientierte Modell stellt für derartig lose gekoppelte Mashup-Systeme eine weitere wissenschaftliche Neuheit dar.
Die Kommunikationskonzepte schließen außerdem Mediationsverfahren auf der Daten- und Schnittstellenebene ein, welche auf den semantischen Abstraktionen des Komponentenmodells aufsetzen. Syntaktische Differenzen zwischen Komponenten können dadurch überbrückt werden – ein Aspekt, der in bisherigen Systemen zur semantischen Suche und Komposition lediglich programmatisch möglich war.

Herausforderung ❸: Adaption

Formalisierung von Adaptionstechniken für universelle Kompositionen

Die in Abschnitt 5.3 vorgestellten Konzepte liefern erstmals eine Lösung, um adaptives Verhalten universeller Komposition in abstrakter Form zu beschreiben, wie in These 4 formuliert. Die aspektorientierte Repräsentation im Kompositionsmodell erlaubt u. a. die anwendungs- und plattformübergreifende Wiederverwendung des Adaptionswissens. Einmal formalisierte, typische Adaptionsbelange, z. B. zur Bildschirm-, Orts- oder Nutzeranpassungen, können so in verschiedenen Anwendungen zum Einsatz kommen, ohne dass sie neu erstellt werden müssen. Context Links bieten darüber hinaus eine intuitive Modellierungsform zur Konfiguration bzw. Synchronisation von Komponenten mit Kontexteigenschaften.
Da sich die konzipierten Adaptionstechniken auf die Abstraktionen des o. g. universellen Komponentenmodells abstützen, bleibt die Adaptionslogik von Änderungen der Komponenten selbst unberührt. Gleichzeitig können Anpassungen über diese universelle Schnittstelle auch auf jene Komponenten angewendet werden, die nicht explizit als adaptiv entwickelt wurden.
Die Adaptionstechniken beschränken sich auch nicht nur auf einzelne Aspekte, wie die Auswahl von Diensten, die Anpassung ausgetauschter Nachrichten oder die Manipulation HTML-basierte Oberflächen, wie in bisherigen Ansätzen. Mit der nahezu beliebigen Anpassung aller Belange des Kompositionsmodells können alle Anwendungsebenen adressiert werden. Neben der Rekonfiguration und den Austausch von Komponenten sind somit auch dynamische Adaptionen der Kompositionsebene, z. B. von Layout, Daten- und Kontrollfluss, möglich. Durch die Abstraktionen des Kompositionsmodells und die Formalisierung der Techniken in Form klar strukturierter Adaptionsaktionen stellt sich das Authoring dennoch einfacher dar, als die bisherige, programmatische Umsetzung.
Neben der Wiederverwendbarkeit und Einfachheit der Abstraktionen liegt ein weiterer Vorteil des Konzeptes in der Reduktion potentieller Fehler: Der Überblick über das adaptive Verhalten einer Anwendung ist auf Basis der Aspekte eher gegeben, als in Quellcode. Fehler und Konflikte können so schneller erkannt und behoben werden.

Kontextualisierung und dynamische Anpassung

Die in Abschnitt 6.3 vorgestellten Konzepte zur Adaption kompositer Webanwendungen verdeutlichen, wie die o. g. Adaptionsaspekte interpretiert und im Zusammenspiel mit verschiedenen Laufzeitumgebungen realisiert werden können. Sie untermauern somit These 4 zur Laufzeit. Die adaptive Anwendung sowie die MREs bleiben dabei von den aufwändigen Belangen der Kontexterfassung, -modellierung und -auswertung befreit. Letztere wurden in dedizierte Dienste wie CroCo ausgelagert, die das Kontextwissen allen Bestandteilen der Kompositionsinfrastruktur zur Verfügung stellen. Es kann somit durch verschiedene Kompositionsplattformen genutzt werden.
Diese Nutzung übernimmt das konzipierte und erfolgreich erprobte Adaptionssystem, welches die o. g. formalisierten Adaptionsaspekte dynamisch auswertet und die Anpassung kompositer Webanwendungen auf potentiell allen Anwendungsebenen nach sich zieht. Die entsprechenden Adaptionstechniken sind folglich nicht mehr auf Web Services und den Nachrichtenaustausch (wie in SOA) oder Hypermedia-Oberflächen (wie im Web Engineering) beschränkt – sie adressieren sowohl die Komponenten- als auch die Kompositionsebene. Gleichzeitig bleibt die Entwicklung der Komponenten selbst von der Adaption unberührt, da die Adaptionstechniken auf die Abstraktionen des universellen Komponentenmodells abgebildet werden.
Das Adaptionssystem stellt die erste Lösung dar, mit der Kontextadaptivität in universell komponierten, interaktiven Webanwendungen unterstützt werden kann, und welches direkt auf deren Charakteristika abgestimmt ist. Durch die Unabhängigkeit der Adaptionsbeschreibung von MREs und bestimmten Technologien bleibt die Auswertung der Adaptionslogik von der Anwendungsausführung entkoppelt – das Adaptionssystem kann deshalb mit verschiedenen MREs zum Einsatz kommen. Dies trägt zu deutlich leichtgewichtigeren und weniger komplexen Kompositionsplattformen bei, die lediglich Implementierungen für die Adaptionsaktionen selbst bereitstellen müssen. Auch bestehende Plattformen können auf diese Weise leicht erweitert werden, um Kontextadaptivität zu realisieren.

Kernbeiträge und Fazit

Entsprechend der obigen Diskussion können die **Kernbeiträge** dieser Arbeit wie folgt zusammengefasst werden:

- Konzeption eines universellen, zustandsbasierten Komponentenmodells zur Repräsentation verteilter Anwendungskomponenten der Daten-, Geschäftslogik- und Präsentationsebene.
- Entwicklung einer deklarativen, technologieunabhängigen Beschreibungssprache zur Bereitstellung und Verwaltung dieser Komponenten.
- Spezifikation eines plattformunabhängigen, belangorientierten Metamodells zur Beschreibung universell komponierter, interaktiver Webanwendungen.
- Konzeption von Algorithmen zur dynamischen, kontextadaptiven Suche, Gewichtung und Auswahl der o. g. Komponenten.
- Entwurf einer modularen, verteilten Kompositionsinfrastruktur zur semantischen Verwaltung, Suche und Bereitstellung von Komponenten.

- Entwicklung einer Referenzarchitektur als Ausführungsumgebung lose gekoppelter, universeller Kompositionen.
- Formalisierung und abstrakte Beschreibung von Adaptionsmethoden und -techniken zur dynamischen Anpassung universell komponierter Anwendungen.
- Entwicklung eines generischen Adaptionssystems zur Unterstützung der o. g. Adaptionstechniken in Verbindung verschiedenen Kompositionsumgebungen.

In Kombination ergeben die Beiträge den ersten ganzheitlichen Ansatz zur modellgetriebenen Entwicklung adaptiver, komponentenbasierter Webanwendungen, der die Vorteile der Serviceorientierung durchgängig für ihre Erstellung, Komposition und Kontextualisierung nutzbar macht. Der angestrebte „leichgewichtige" Erstellungsprozess, der bislang statischen Mashups der Daten- und Anwendungslogikebene vorbehalten war, wurde somit erfolgreich auf interaktive Webanwendungen übertragen. Die Praktikabilität und Tauglichkeit der vorgestellten Beiträge wurde durch eine umfangreiche prototypische Implementierung und anhand diverser Beispielanwendungen hinreichend erprobt und validiert.
Nicht alle Thesen aus Abschnitt 1.1.2 – insbesondere die qualitativen Aussagen hinsichtlich der Beschleunigung und Vereinfachung im Entwicklungsprozess – können an dieser Stelle abschließend bewertet werden, da sie teilweise umfassende, langfristige Studien nötig machen. Die vorliegenden Konzepte und ihre prototypische Umsetzung bieten hierfür allerdings eine fundierte Grundlage. Über die Grenzen der entwickelten Konzepte sowie mögliche Ansatzpunkte zur Weiterentwicklung geben die folgenden Abschnitte Aufschluss.

8.2.2 Einschränkungen und Grenzen

Die vorgestellten Konzepte unterliegen einigen Grundannahmen, die die Anwendbarkeit und den Einsatz der geschaffenen Lösung einschränken. Diese Einschränkungen sind nicht zwingend konzeptioneller Natur, sondern bestehen aufgrund des Abgrenzung von Randthemen und der zeitlichen Begrenzung der Arbeit. Für die Mehrzahl der im folgenden genannten Aspekte wurden deshalb im Konzept Erweiterungspunkte geschaffen. Einige davon werden bereits im Rahmen angeschlossener und Folgeprojekte behandelt, andere bieten Ansatzpunkte für zukünftige Arbeiten und werden genauer in Abschnitt 8.3 skizziert.

Einschränkungen im Autorenprozess

Das Modellierungskonzept setzt zur funktionalen Annotation und Typisierung von Komponenten auf gemeinsame, semantische Konzeptualisierungen. Insbesondere bei der Kapselung bestehender Dienste und UI-Komponenten ist diese gemeinsame semantische Basis nicht zwingend gegeben. Deshalb müssen Verfahren zur Äquivalenzbildung bzw. Abbildung entwickelt und integriert werden, die den Einsatz verschiedener semantischer Domänenmodelle erlauben.
Zur Vereinfachung des Autorenprozesses wurde bewusst die Entscheidung getroffen, bei der Modellierung auf komplexe Kontrollflusskonstrukte zu verzichten. Umfangreiche Workflows und Transaktionen können deshalb nur teilweise auf das Kompositionsmodell abgebildet werden. Hier muss stattdessen die Verknüpfung mit bestehenden Systemen erfolgen, wie am Beispiel der HT-MRE (vgl. Abschnitt 7.2.2.3) exemplarisch gezeigt wurde.

Der Aspekt des *Look-and-Feel* findet im Kompositionsmodell bislang schwerpunktmäßig hinsichtlich der visuellen Gestaltung der UI-Komponenten Beachtung. Durch Styles kann beispielsweise das homogene Aussehen der Anwendungsoberfläche sichergestellt werden. Nur bedingt (über Ranking-Regeln) kann bislang hingegen darauf Einfluss genommen werden, dass integrierte Komponenten gleichartige Interaktionstechniken und -metaphern nutzen. Zur Steigerung der Gebrauchstauglichkeit der kompositen Anwendungen sollte dieser Punkt adressiert werden.

Schließlich wurde mit dem grafischen Modelleditor ein einfaches Werkzeug geschaffen, um die Erstellung und Validierung von Anwendungsmodellen zu ermöglichen. Die Tauglichkeit des Editors für Endnutzer ist jedoch als begrenzt einzuschätzen und stand nicht im Fokus der Arbeit. Für die nutzergetriebene, universelle Komposition müssen deshalb adäquate, visuelle Autorenwerkzeuge geschaffen und deren Tauglichkeit im Rahmen von Nutzerstudien evaluiert werden.

Einschränkungen bei der Komposition und Ausführung

Die Suche, Auswahl und Integration von Komponenten zur Laufzeit führt zu einem Mehraufwand, der sich in einer, wenn auch vertretbaren, zeitlichen Verzögerung bei der Auslieferung von Anwendungen äußert. Die Entwicklung ausgereifter Mechanismen zur Performanzsteigerung stand nicht im Mittelpunkt der Arbeit und wurde deshalb nur rudimentär adressiert.

Bei der Suche bzw. beim Matching von Komponenten wird zur Reduktion der Komplexität bislang nur die direkte Abbildung zwischen Bestandteilen der semantischen Vorlage und Komponenten betrachtet. Es sind jedoch auch 1:n- und n:m-Beziehungen denkbar, bei denen beispielsweise mehrere Kandidaten genau einer semantischen Vorlage des Anwendungsmodells entsprechen. Derartige Zusammenhänge werfen jedoch neue Forschungsprobleme auf, die im Rahmen zukünftiger Arbeiten behandelt werden müssen. Mit der Möglichkeit der hierarchischen Komposition wurde bereits ein erster Lösungsansatz skizziert: So kann jede Komposition wiederum als Komponente repräsentiert und im Rahmen der Discovery eingebunden werden.

Eine weitere Herausforderung besteht in der Überschneidung der semantischer Annotationen von Komponenten. Das Konzept geht bislang von eindeutig typisierten Schnittstellen aus, sodass im Matching zweifelsfrei zwischen verschiedenen Properties, Operationen und Ereignissen unterschieden werden kann. Grundlage bildet die Nutzung einheitlicher semantischer Konzepte. Mit der oben motivierten Nutzung verschiedener Domänenmodelle steigt jedoch die Wahrscheinlichkeit für Überdeckungen hinsichtlich des Datentyps oder der Funktionalität. Für diesen Fall muss der Matching-Algorithmus konzeptionell erweitert werden.

Die korrekte Ausführung der komponierten Anwendungen ist stark von der Robustheit und Fehlertoleranz der jeweils integrierten Komponenten abhängig. Die Komponentenentwicklung wurde in dieser Arbeit jedoch nicht explizit betrachtet. Über die Erfüllung der Schnittstelle zur MRE hinaus müssen deshalb Konzepte entwickelt werden, die das Testen und die Qualitätssicherung von Komponenten aktiv unterstützen. Gleichermaßen bedarf Fehlerbehandlung und Qualitätssicherung durch die Laufzeitumgebungen weiterer konzeptioneller Betrachtung, da sie in den vorgestellten Konzepten und Umsetzungen nur in Ansätzen unterstützt werden.

Die Tauglichkeit und Praktikabilität der Laufzeitkonzepte wurde anhand diverser Beispielanwendungen geprüft. Eine ausführliche Evaluation macht jedoch eine

ausreichend große Komponentenbasis und Tests im realen Anwendungskontext nötig. Eine abschließende Bewertung im Rahmen deutlich umfangreicherer Anwendungen steht deshalb noch aus.

Einschränkungen der Adaptionskonzepte

Die Formulierung der Adaptionsaspekte wird, wie die Anwendungsmodellierung, durch den angesprochenen, grafischen Editor ermöglicht. Hier ist eine stärkere Autorenunterstützung und Analytik anzustreben, die u. a. Aspekte vorschlagen, Konflikte und Wechselwirkungen zwischen ihnen erkennen, dem Autor präsentieren und ggf. lösen kann.
Wie bereits dargestellt wurde, decken die formalisierten Adaptionsaktionen alle typischen Adaptionsszenarien, wie die Personalisierung und Endgeräteanpassung ab. Voraussetzung hierfür ist das Vorhandensein aktueller, konsistenter Kontextparameter, die bei der Erprobung und Validierung der Konzepte als gegeben angenommen wurden. Neben den vorgestellten müssen somit ggf. weitere Kontextmonitoren und -sensoren entwickelt und dem Adaptionssystem hinzugefügt werden, um die gewünschten Anpassungen auslösen zu können.
Die Adaptionskonzepte umfassen Anpassungen auf der Komponenten- und Kompositionsebene. Sie beschränken sich aber auf die Anwendung selbst, d. h. ihre Bestandteile, inhaltlichen Zusammenhänge und Abstraktionen im Kompositionsmodell. Weitergehende Adaptionsmethoden, z. B. zur Adaption der Systemarchitektur bzw. der dynamischen Verteilung zwischen Client und Server wurden nicht vorgesehen. Diese sind nur bedingt mit der plattformunabhängigen Modellierung vereinbar, da die Formulierung entsprechender Aspekte das Wissen über die Verortung von Komponenten nötig macht. Deshalb müssen ggf. transparente Verteilungsmechanismen entwickelt werden.

8.3 Laufende und zukünftige Arbeiten

Die in dieser Arbeit geschaffenen Konzepte und ihre praktische Realisierung bieten Anknüpfungspunkte zur Behandlung einer Vielzahl weiterführender Forschungsfragen. Einige davon werden bereits im Rahmen angeschlossener Forschungsprojekte adressiert, für die die vorliegende Arbeit sowohl die konzeptionelle als auch die praktische Grundlage darstellt. Auf sie wird im Folgenden an entsprechender Stelle verwiesen. Weiterhin haben die entwickelten Lösungen auch Einfluss auf die industrielle Praxis. So fließen die Erkenntnisse u. a. in die Entwicklung der PaaS-Lösung CAS Open [@Niemann, 2009] der CAS Software AG ein, die in das angeschlossene Forschungsprojekt eingebunden war.
Die folgenden Punkte verdeutlichen gleichsam die Herausforderungen wie auch das Potential der geschaffenen Lösungen anhand der wichtigsten und interessantesten Fragestellungen für weiterführende Arbeiten.

Integration mit bestehenden Entwicklungsprozessen und Plattformen

In dieser Arbeit wurde ein innovativer Lösungsansatz geschaffen, der die Entwicklung und Wartung individueller Anwendungslösungen vereinfacht und eine Vielzahl bereits genannter Vorteile verspricht. Trotz der Offenheit und Nutzung von Standards bei der Konzeption und Umsetzung müssen für den Einsatz im geschäftlichen

Umfeld jedoch sowohl konzeptionelle als auch praktische Rahmenbedingungen geschaffen werden, um den Ansatz mit bestehenden Entwicklungsprozessen und Softwareplattformen zu verknüpfen.

So ist beispielsweise die Kopplung mit EMML-Mashups (vgl. Abschnitt 3.2.2.2) denkbar, welche durch ein breites Industriekonsortium getragen werden. Sie beschränken sich auf der Verknüpfung und Manipulation von Daten über relationale Operationen und können somit als Komponenten repräsentiert und integriert werden. Dafür bedarf es jedoch neuer, technischer Konzepte, welche u. a. die Kopplung von MREs und EMML-kompatiblen Plattformen erlauben. Gleichermaßen gilt es zu untersuchen, wie andere Komponententypen bzw. -modelle möglichst transparent auf das vorliegende Konzept abgebildet werden können. Für Web Services wurden dies bereits im Rahmen der Arbeit erprobt. Zur An- und Einbindung anderer Dienste, Ressourcen oder Widgets bedarf es allerdings weiterer Forschungsarbeit.

Weiterhin sollte untersucht werden, wie bereits etablierte Vorgehensmodelle und Prozesse als Grundlage für die universelle Komposition genutzt werden können. In diesem Zusammenhang widmet sich das angeschlossene Projekt DEMISA [@DEMISA, 2011] der Frage, wie im Unternehmen bereits formalisierte Anforderungen und Abläufe in Form von Geschäftsprozessen semi-automatisch auf den vorgestellten Kompositionsansatz abgebildet werden können.

Verbesserte Autorenunterstützung und End User Development

Wie angestrebt, unterstützen die Abstraktionen des Komponenten- und Kompositionsmodells die Entwicklung kompositer Anwendungen durch Nicht-Programmierer. Somit können situative Lösungen direkt durch Domänenexperten erstellt werden. Diese benötigen jedoch ausreichend mächtige und gleichzeitig nutzerfreundliche Autorenwerkzeuge, die ausgehend vom bestehenden Modelleditor bzw. dessen API entwickelt und in Nutzerstudien evaluiert werden müssen.

Langfristig zeichnet sich die Verschmelzung von Entwicklungs- und Laufzeit ab, d. h. bestehende, komposite Anwendungen können während ihrer Ausführung direkt verändert und erweitert werden. Dies macht Mechanismen zur Kontexterfassung, -auswertung und zur proaktiven Empfehlung neuer Komponenten nötig. Im angelagerten Forschungsprojekt EDYRA [@EDYRA, 2011] werden entsprechende wissenschaftliche Konzepte auf Basis der bestehenden Infrastruktur entwickelt.

Erweiterung der semantischen Discovery

Die geschaffenen Konzepte zur semantische Suche, Gewichtung und Auswahl von Komponenten zur Laufzeit berücksichtigen bislang neben der funktionalen Passgenauigkeit zwei Aspekte: Einerseits werden implizit plattformspezifische Ausschlusskriterien, z. B. hinsichtlich der technologischen Kompatibilität und Verfügbarkeit benötigter Plug-ins, berücksichtigt, andererseits die expliziten, kontextabhängigen Rankingregeln interpretiert. Dieser Suchprozess kann durch verschiedene Strategien erweitert und verbessert werden:

Zum einen kann die Suche durch die Kombination mit nicht-logikbasierten Matching-Verfahren optimiert werden. Ein typisches Beispiel hierfür ist die Einbeziehung der Namen von Properties, Operations und Events in den Abgleich unter Nutzung von Thesauri und Wörterbüchern. Die Zuordnung zwischen Vorlagen und Kandidaten kann so gerade bei Uneindeutigkeiten zweifelsfrei erfolgen.

Zum anderen verspricht die Nutzung von Domänenwissen, z. B. welche Interaktionstechnik für welches Endgerät bzw. welche Situation geeignet ist, eine bessere Passgenauigkeit der ausgelieferten Komponenten. Dazu muss es formalisiert und konzeptionell geklärt werden, wie es den Gewichtungs- und Auswahlprozess beeinflusst. Am Beispiel der bedarfsgerechten Bereitstellung von Visualisierungskomponenten für semantische Daten werden diese Ziele im Projekt VizBoard [Voigt et al., 2012b] auf Basis der vorgestellten Konzepte verfolgt. Auch die Einbeziehung kollaborativen Wissens („*Nutzer X hat in ähnlichem Kontext Komponente Y genutzt und sie gut bewertet*“) besitzt ein hohes Potential an innovativen Lösungen, aber auch an bislang unbeantworteten Forschungsfragen.

Schließlich besteht eine weitere, offene Herausforderung in der Unterstützung verschiedener, semantischer Domänenmodelle: Im Integrationsprozess müssen dazu Ontology-Matching bzw. -Mapping-Verfahren vorgesehen werden, die die semantischen Konzepte verschiedener Komponentenbeschreibungen aufeinander bzw. auf ein gemeinsames Grounding abbilden können. Die Konzepte müssen auch die automatische Mediation zur Laufzeit abdecken.

Mechanismen für Qualitätssicherung und Datensicherheit

Die Entwicklung von Anwendungen durch Nicht-Programmierer macht umfassende Maßnahmen zur Qualitätssicherung nötig. Domänenexperten können zwar die Korrektheit einer Komposition aus fachlicher Sicht einschätzen, ihnen fehlt jedoch i. d. R. das Verständnis bestimmter programmatischer Zusammenhänge. Deshalb benötigen sie bereits bei der Modellierung Führung, um potentielle Fehler, z. B. redundante Kommunikation, Zirkelbezüge oder widersprüchliche Trigger, zu vermeiden. Die Validierungsmechanismen im Modell müssen dementsprechend erweitert und mit Konzepten der aktiven Autorenunterstützung angereichert werden.

Besonders im geschäftlichen Umfeld muss die Qualität von Anwendungen und deren Komponenten auch zur Laufzeit überwacht und sichergestellt sein. Ein typisches Beispiel hierfür ist die Einhaltung minimaler Antwortzeiten durch Komponenten bzw. Dienste und die Gesamtanwendung. Hierfür ist die Erweiterung des Kompositionsmodells um QoS-Kriterien vorstellbar, die mit den Zusicherungen und überwachten, tatsächlichen Werten von Komponenten und Kompositionen permanent abgeglichen werden. Dabei kann auf die umfangreichen Erfahrungen und Konzepte der Web-Service-Orchestrierung zurückgegriffen werden – diese müssen jedoch konzeptionell erweitert werden, um u. a. die Besonderheiten der Präsentationsebene zu berücksichtigen.

Die dynamische Bindung der Anwendungsbestandteilen wirft weiterhin Fragen der Datensicherheit auf, z. B. welche Komponenten von welchem Anbieter auf welche Daten der Anwendung Zugriff erhalten, und wie sie damit umgehen dürfen. Beispielsweise sollte vermieden werden, dass UI-Komponenten von Drittanbietern angezeigte Daten im Hintergrund an fremde Server senden, falls dies nicht ausdrücklich gewünscht ist. Deshalb sollten Rahmenbedingungen geschaffen werden, die u. a. die Zertifizierung von Komponenten, die Verschlüsselung des Datenaustauschs und grundlegende Authentifizierungsmechanismen unterstützen. Auch die Definition von Zugriffsrechten und die Überwachung des Datenverkehrs sind zur Steigerung der Informationssicherheit denkbar.

Sicherstellung und Evaluation der Gebrauchstauglichkeit

Die Vorteile und Potentiale der dynamischen Bindung aller Anwendungsbestandteile wurden hinlänglich erläutert. Für Designer und Usability-Experten sind sie jedoch ein Problem, da diese die Anwendungsoberfläche zum Entwicklungszeitpunkt nicht sehen und nur über Abstraktionen beeinflussen können. Der Wunsch, möglichst konkrete Oberflächen zu entwerfen, steht hier der nötigen Flexibilität für den Einsatz in verschiedensten Kontexten gegenüber. Da die Gebrauchstauglichkeit der geschaffenen Lösungen ein zentrales Akzeptanzkriterium seitens der Nutzer darstellt, müssen Konzepte entwickelt werden, um beiden Ansprüchen gerecht zu werden.

Einerseits bedarf es der Unterstützung von Usability-Kriterien im Entwicklungsprozess. Dies kann von der Einbeziehung von *Best Practises* hinsichtlich der Konfiguration von Komponenten bis hin zur Simulation der komponierten Anwendung in vordefinierten Ausführungskontexten reichen. Neben der Umsetzung entsprechender Autorenwerkzeuge sind dafür aber Erweiterungen am Komponentenmodell nötig, um die Auszeichnung mit Usability-relevanten Eigenschaften zu ermöglichen. Andererseits kann Usability auch zur Kompositionszeit Beachtung finden. Die bereits dargestellte Einbeziehung von Domänenwissen in den Auswahlprozess von Komponenten ist eine solche Möglichkeit: Formalisiertes Wissen um die Gebrauchstauglichkeit von Anwendungen, z. B. bezüglich visueller und Interaktionsmetaphern, kann hier aktiv zur Auswahl möglichst gebrauchstauglicher Komponenten beitragen.

Weiterentwicklung der Adaptionsbelange

Mit den abstrakten und wiederverwendbaren Adaptionsaspekten wurde in dieser Arbeit der Grundstein zur reichhaltigen Anpassung kompositer Anwendungen zur Laufzeit gelegt. Die Beschreibung des adaptiven Verhaltens wurde zwar vereinfacht und durch das Adaptionsmodell formalisiert – für Domänenexperten und Endanwender ist sie dennoch nur bedingt geeignet. Neben der besseren Unterstützung durch Autorenwerkzeuge, wie bereits oben angedeutet, ist in diesem Zusammenhang die weitere Abstraktion und Zusammenfassung von Aspekten zu höherwertigen Adaptionsbelangen anzustreben. Die Nutzung derartiger „Muster“, wie *Bildschirmanpassung*, *Ortsanpassung* und *Internationalisierung*, würde den Autorenprozess weiter vereinfachen und potentielle Konflikte zwischen Aspekten könnten bereits im Vorfeld ausgeschlossen bzw. einfacher erkannt werden.

Die Erkennung und Behandlung derartiger Konflikte bzw. „Interaktionen“ ist ein weiteres Forschungsproblem, welches ausgehend von den vorgestellten Konzepten adressiert werden sollte. So kann es zwischen Adaptionsaspekten hinsichtlich der Trigger, Bedingungen und insbesondere der auszuführenden Anpassungen zu Widersprüchen und unerwünschten Effekten kommen. Im schlechtesten Fall wird ein Aspekt durch einen anderen verdeckt oder konterkariert. Für die Analyse und Vermeidung derartiger Konflikte müssen neue wissenschaftliche Konzepte entwickelt werden. Hierzu kann u. a. auf Erfahrungen aus dem HyperAdapt-Projekt [HyperAdapt, 2011] aufgebaut werden.

Anhänge

A.1 Komponentenbeschreibung in SMCDL

Codebeispiel A.1 zeigt die Beschreibung der Komponente *RoutingMap* in Auszügen, die in der Anwendung *TravelMash* zum Einsatz kommt.

```
<?xml version="1.0" encoding="UTF-8"?>
<component xmlns="http://inf.tu-dresden.de/cruise/mcdl" name="RoutingMap"
  id="http://... /ui/maps/ExtRoutingMap" version="1.1" type="ui"
  functionality="act:Zoom act:RouteVisualization ..."
  xmlns:meta="..." xmlns:nfp="..." xmlns:travel="..." etc.>
 <documentation>A map that visualizes ... </documentation>
 <meta:metadata>
  <meta:keywords>map start destination route, google, ...</meta:keywords>
  <meta:screenshots> ... </meta:screenshots>
  ... <!-- additional metdata -->
 </meta:metadata>
 <interface>
  <!-- properties -->
  <property name="title" type="mcdl:hasTitle" required="true"/>
  <property name="mapType" type="viso:MapType"/>
  <property name="currentLocation" type="travel:Location"/>
  <!-- events -->
  <event name="startChanged" trigger="interaction"
    dependsOn="act:InputStartLocation" functionality="evt:StartLocationUpdated">
    <parameter name="location" type="travel:Location"/>
  </event>
  <event name="destChanged" trigger="interaction"
    dependsOn="act:InputDestLocation" functionality="evt:DestLocationUpdated">
    <parameter name="location" type="travel:Location"/>
  </event>
  <!-- operations -->
  <operation name="setMarker" functionality="act:OutputLocation">
    <parameter name="locationtype" type="travel:locationType"/>
    <parameter name="location" type="travel:Location"/>
  </operation>
  ... <!-- additional interface artefacts -->
 </interface>
 <binding bindingtype="default">
  <dependencies>
   <dependency uri="http://... /ext/resources/css/ext-all.css" type="text/css"/>
   <dependency uri="http://maps.google.com/maps/api/js" type="text/javascript"/>
   <dependency uri="http://.../maps/ExtRoutingMap/map.js" type="text/javascript"/>
  </dependencies>
  <constructor>
   <code>new StartDestinationMap()</code>
  </constructor>
  ... <!-- additional binding information -->
 <binding>
</component>
```

Lst. A.1: Beispiel einer Komponentenbeschreibung in SMCDL

A.2 Komponentenmodell in Form der MCDO

Codebeispiel A.2 zeigt einen Auszug der Serialisierung einer Komponentenbeschreibung in RDF/XML auf Basis des Vokabulars der MCDO. Einige URLs wurden durch die Angabe von [[*shorturl*]] der Übersichtlichkeit halber gekürzt.

```
<rdf:RDF xmlns:rdf="..." xmlns:owl="..." xmlns:rdfs="..." xmlns:xsd="..."
    xmlns:mcdo="[[shorturl]]/mcd.owl#" xmlns:nfp="[[shorturl]]/nfp.owl#" ...>
  <rdf:Description rdf:about="http://vvo-online.de/services/routing/vvo">
    <rdf:type rdf:resource="[[shorturl]]/mcd.owl#MashupComponent"/>
    <mcdo:hasId>http://vvo-online.de/services/routing/vvo</mcdo:hasId>
    <mcdo:isUI rdf:datatype="[[shorturl]]/XMLSchema#boolean">false</mcdo:isUI>
    <mcdo:hasName>vvoService</mcdo:hasName>
    <mcdo:hasCapability rdf:resource="[[shorturl]]/services/routing"/>
    <mcdo:hasMetadata rdf:nodeID="A4"/>
    <mcdo:hasInterface rdf:nodeID="A10"/>
    <mcdo:hasBinding rdf:nodeID="A11"/>
  </rdf:Description>
   <rdf:Description rdf:about="[[shorturl]]/services/routing">    <!-- Capability -->
    <rdf:type rdf:resource="[[shorturl]]/mcd.owl#Capability"/>
    <mcdo:hasFunctionality>[[shorturl]]/actions.owl#RouteVisualization</mcdo:
        hasFunctionality>
    <mcdo:hasEntity>[[shorturl]]/travel.owl#RouteList</mcdo:hasEntity>
    <mcdo:hasProvider>system</mcdo:hasProvider>
  </rdf:Description>
   <rdf:Description rdf:nodeID="A4">                              <!-- Metadata -->
    <rdf:type rdf:resource="[[shorturl]]/nfp.owl#MetaData"/>
    <nfp:hasEnergyConsumption rdf:resource="[[shorturl]]/nfp.owl#Low"/>
    <nfp:hasPricing rdf:nodeID="A7"/>
    <nfp:hasScreenShot rdf:nodeID="A8"/>
    <nfp:hasKeyword rdf:datatype="[[shorturl]]/XMLSchema#string">public transport VVO
        </nfp:hasKeyword>
  </rdf:Description>
    <rdf:Description rdf:nodeID="A10">                           <!-- Interface -->
    <rdf:type rdf:resource="http://[[shorturl]]/mcd.owl#Interface"/>
    <mcdo:hasOperation rdf:nodeID="A1"/>
    <mcdo:hasEvent rdf:nodeID="A14"/>
    <mcdo:hasProperty rdf:nodeID="A15"/>
    <mcdo:hasProperty rdf:nodeID="A16"/>
  </rdf:Description>
  <rdf:Description rdf:nodeID="A11">                             <!-- Binding -->
    <rdf:type rdf:resource="http://[[shorturl]]/mcdl.owl#Binding"/>
    <mcdo:hasBindingType rdf:datatype="[[shorturl]]/XMLSchema#string">default</mcdo:
        hasBindingType>
    <mcdo:hasConstructor rdf:nodeID="A21"/>
    <mcdo:hasRuntime rdf:nodeID="A19"/>
    <mcdo:hasDependency rdf:nodeID="A22"/>
    <mcdo:hasDependency rdf:nodeID="A13"/>
      </rdf:Description>
    <rdf:Description rdf:nodeID="A1">                            <!-- Operation -->
    <rdf:type rdf:resource="[[shorturl]]/mcd.owl#Operation"/>
    <mcdo:hasName rdf:datatype="[[shorturl]]/XMLSchema#string">calculateRoutes</mcdo:
        hasName>
    <mcdo:hasCallbackEvent rdf:nodeID="A2"/>
    <mcdo:hasParameter rdf:nodeID="A3"/>
  </rdf:Description>
  <rdf:Description rdf:nodeID="A13">                             <!-- Dependency -->
    <rdf:type rdf:resource="[[shorturl]]/mcd.owl#Dependency"/>
    ...
  </rdf:Description>
  <!-- many more nodes... -->
</rdf:RDF>
```

Lst. A.2: Beispiel der ontologiebasierten Komponentenbeschreibung

A.3 Kompositionsmodell in EMF

Das folgende Codebeispiel zeigt einen Auszug des Kompositionsmodells der in Abschnitt 7.3.1 vorgestellten kompositen Anwendung *TravelMash*.

```
<?xml version="1.0" encoding="UTF-8"?>
<mcm:MashupComposition xmi:version="2.0" xmlns:xmi="..." xmlns:xsi="..."
    xmlns:mcm="http://../compositionmodel/1.8" name="TravelMash">
  <conceptualModel>
    <styles> ... </styles>
    <environment> ... </environment>
    <components>
      <component xsi:type="mcm:ServiceComponent" name="HotelDienst"
        id="[[shorturl]]/services/hotels/kayak">...</component>
      <component xsi:type="mcm:UIComponent" name="Kalender"
        id="http://.../DatePicker" isResizable="false">
        <event name="updateStartTime">
          <parameter type="[[shorturl]]/travel.owl#hasStartTime" name="startTime"/>
        </event>
        <property type="[[shorturl]]/travel.owl#hasDestinationTime" name="destTime"/>
        <property type="[[shorturl]]/travel.owl#hasStartTime" name="startTime"/>
        ...
      </component>
      <component xsi:type="mcm:UIComponent" name="WetterAnzeige" isTemplate="true">
        <operation name="update">
          <parameter type="[[shorturl]]/travel.owl#Location" name="location"/>
          <functionality URI="[[shorturl]]/travelPlanning.owl#OutputWeather"/>
        </operation>
        ...
        <property type="[[shorturl]]/travel.owl#Location" name="location"/>
      </component>
    </components>
    <rankingRules>
      <rankingRule xsi:type="mcm:OccuranceRule" id="maxPrice" metadata="..." />
      ...
    </rankingRules>
  </conceptualModel>
  <layoutModel>
    <layout xsi:type="mcm:AbsoluteLayout" name="full-fixed">
      <bounds height="750" width="1300" unit="pixel"/>
      <position locate="Kalender" x="0" y="0" unit="pixel">
        <bounds height="160" width="220" unit="pixel"/>
      </position>
      <position locate="WetterAnzeige" x="0" y="160" unit="pixel">
        <bounds height="160" width="220" unit="pixel"/>
      </position>
      ...
    </layout>
  </layoutModel>
  <screenflowModel initialView="Home">
    <view name="Home" vLayout="full-fixed"/>
  </screenflowModel>
  <communicationModel>
    <link xsi:type="mcm:Link" name="DestinationLink">
      <parameter type="[[shorturl]]/travel.owl#Location" name="destination"/>
      <parameter type="[[shorturl]]/travel.owl#locationType" name="type"/>
      <publisher event="//@conceptualModel/@components/
        @component[name='EventAnzeige']/@event[name='eventSelected']"/>
      <subscriber operation="//@conceptualModel/@components/
        @component[name='RoutenAnzeige']/@operation[name='markLocation']"/>
    </link>
    <link xsi:type="mcm:PropertyLink" name="destinationSync">
      <parameter type="[[shorturl]]/travel.owl#Location" name="location"/>
      <participant property="//@conceptualModel/@components/@component[name='
          WetterAnzeige']/@property[name='location']"/>
```

```
60          <participant property="//@conceptualModel/@components/@component[name='
                HotelDienst']/@property[name='location']"/>
61          ...
62        </link>
63        <link xsi:type="mcm:BackLink" name="HotelBackLink">
64          <parameter type="[[shorturl]]/travel.owl#HotelList" name="hotels"/>
65          <requestor event="//@conceptualModel/@components/@component[name='HotelAnzeige
                ']/@event[name='location']"/>
66          <replier operation="//@conceptualModel/@components/@component[name='HotelDienst
                ']/@operation[name='getHotels']"/>
67        </link>
68      </communicationModel>
69    </mcm:MashupComposition>
```

Lst. A.3: Beispiel eines Kompositionsmodells in XMI

Verzeichnis eigener Publikationen

Dachselt, R., Hinz, M. und Pietschmann, S. (Apr. 2006): „Using the AMACONT Architecture for Flexible Adaptation of 3D Web Applications“. In: *Proceedings of the 11th International Conference on 3D Web Technology (Web3D)*. Columbia, Maryland, USA: ACM, S. 75–84. ISBN: 1-59593-336-0.

Hinz, M., Pietschmann, S. und Fiala, Z. (Okt. 2006): „A Framework for Context Modeling in Adaptive Web Applications“. In: *Proceedings of the International IADIS WWW/Internet 2006 Conference*. Murcia, Spanien.

Hinz, M., Pietschmann, S. und Fiala, Z. (Juni 2007a): „A Framework for Context Modeling in Adaptive Web Applications“. In: *IADIS International Journal of WWW/Internet* 5.1.

Hinz, M., Pietschmann, S., Umbach, M. und Meißner, K. (Jan. 2007b): „Adaptation and Distribution of Pipeline-Based Context-Aware Web Architectures“. In: *Proceedings of the Sixth Working IEEE/IFIP Conference on Software Architecture (WICSA)*. Mumbai (Bombay), Indien. ISBN: 0-7695-2744-2.

Liebing, C., Braun, I., Pietschmann, S. und Schill, A. (Sep. 2009): „Towards Development of Web Applications Based on User Interface Services – A Requirement Analysis“. In: *Proceedings of the 19th International Crimean Conference on Microwave & Telecommunication Technology (CriMiCo)*. Sewastopol, Ukraine.

Lorz, A., Rümpel, A., Radeck, C., Blichmann, G. und Pietschmann, S. (Dez. 2011): „Introducing the EDYRA Vision: Engineering of Do-It-Yourself Rich Internet Applications“. In: *Proc. of the IADIS International Conference on Internet Technologies & Society*. Poster Paper.

Mitschick, A., Pietschmann, S. und Meißner, K. (Feb. 2010): „An Ontology-Based, Cross-Application Context Modeling and Management Service“. In: *International Journal on Semantic Web and Information Systems (IJSWIS)*.

Nauerz, A., Pietschmann, S. und Pietzsch, R. (Sep. 2007): „Collaborative Annotation-Driven Adaptation in Web Portals“. In: *Proceedings of the 18th ACM Conference on Hypertext and Hypermedia*. Manchester, UK. ISBN: 978-1-59593-820-6.

Nauerz, A., Pietschmann, S. und Pietzsch, R. (Juli 2008a): „Social Recommendation and Adaption in Web Portals“. In: *Proceedings of the Workshop on Adaptation for the Social Web at AH2008*.

Nauerz, A., Pietschmann, S. und Pietzsch, R. (Juli 2008b): „Using Collective Intelligence for Adaptive Navigation in Web Portals“. In: *Proceedings of the 3rd International Workshop on Adaptation and Evolution in Web Systems Engineering (AEWSE'08)*. Yorktown Heights, New York, USA.

Niederhausen, M., Pietschmann, S., Ruch, T. und Meißner, K. (Apr. 2010). "Web-Based Support By Thin-Client Co-Browsing". In: Badr, Y., Chbeir, R., Abraham, A. und Hassanien, A.-E. (Hrsg.): *Emergent Web Intelligence: Advanced Semantic Technologies* Bd. XVI. 544. Springer Berlin/Heidelberg. isbn: 978-1-84996-076-2.

Pietschmann, S. (2009): „A Model-Driven Development Process and Runtime Platform for Adaptive Composite Web Applications". In: *International Journal on Advances in Internet Technology* 2.4, S. 277–288. issn: 1942-2652.

Pietschmann, S. (2011). "Technologische Entwicklungen und Potentiale zur Unterstützung virtueller Organisationen". In: Benkhoff, B., Engelien, M., Meissner, K. und Richter, P. (Hrsg.): *Erfolgreiche virtuelle Organisationen: Durch Frühwarnung Risiken vermeiden* Bd. 1. 1. Kohlhammer-Verlag.

Pietschmann, S., Mitschick, A., Winkler, R. und Meißner, K. (Dez. 2008a): „CroCo: Ontology-Based, Cross-Application Context Management". In: *Proceedings of the 3rd International Workshop on Semantic Media Adaptation and Personalization*. Prag, Tschechische Republik: IEEE CPS. isbn: 978-0-7695-3444-2.

Pietschmann, S., Nestler, T. und Daniel, F. (Nov. 2010a): „Application Composition at the Presentation Layer: Alternatives and Open Issues". In: *Proceedings of the 12th International Conference on Information Integration and Web-based Applications and Services (iiWAS)*. ACM. isbn: 978-1-4503-0421-4.

Pietschmann, S., Niederhausen, M., Ruch, T., Wilkowski, R. und Richter, J. (Okt. 2007): „Instant Collaborative Web-Browsing with VCS". In: *Virtuelle Organisationen und Neue Medien (GeNeMe 2007)*. Dresden, Deutschland.

Pietschmann, S., Radeck, C. und Meißner, K. (Sep. 2011a): „Facilitating Context-Awareness in Composite Mashup Applications". In: *Proceedings of the 3rd International Conference on Adaptive and Self-Adaptive Systems and Applications (ADAPTIVE)*. XPS. isbn: 978-1-61208-011-6.

Pietschmann, S., Radeck, C. und Meißner, K. (Sep. 2011b): „Semantics-Based Discovery, Selection and Mediation for Presentation-Oriented Mashups". In: *Proceedings of the 5th International Workshop on Web APIs and Service Mashups*. New York, NY, USA: ACM. isbn: 978-1-4503-0823-6.

Pietschmann, S. und Tietz, V. (Okt. 2008): „Anwendungsübergreifende Web-2.0-Kollaborationsmuster". In: Meissner, K. und Engelien, M. (Hrsg.): *Virtuelle Organisationen und Neue Medien (GeNeMe 2008)*. Dresden, Deutschland.

Pietschmann, S., Tietz, V. und Meißner, K. (Juli 2008b): „Cross-Application, Pattern-Based Web Collaboration". In: *Proceedings of the International Conference on Web-Based Communities (WBC)*. Amsterdam, Niederlande.

Pietschmann, S., Tietz, V., Reimann, J., Liebing, C., Pohle, M. und Meißner, K. (Nov. 2010b): „A Metamodel for Context-Aware Component-Based Mashup Applications". In: *Proceedings of the 12th International Conference on Information Integration and Web-based Applications & Services (iiWAS)*. ACM. isbn: 978-1-4503-0421-4.

Pietschmann, S., Voigt, M. und Meißner, K. (Okt. 2009a): „Adaptive Rich User Interfaces for Human Interaction in Business Processes". In: *Proceedings of the 10th International Conference on Web Information Systems Engineering (WISE)*. WISE. Posen, Polen: Springer Berlin/Heidelberg. isbn: 978-3-642-04408-3.

Pietschmann, S., Voigt, M. und Meißner, K. (Mai 2009b): „Dynamic Composition of Service-Oriented Web User Interfaces". In: *Proceedings of the 4th International*

Conference on Internet and Web Applications and Services (ICIW). Mestre/Venedig, Italien: IEEE CPS, S. 217–222. ISBN: 978-0-7695-3613-2.

Pietschmann, S., Voigt, M. und Meißner, K. (Juli 2012): „Rich Communication Patterns for Mashups“. In: *Proc. of the 12th International Conference on Web Engineering (ICWE 2012)*. LNCS Bde. 7387. Springer, S. 315–322.

Pietschmann, S., Voigt, M., Rümpel, A. und Meißner, K. (Juni 2009c): „CRUISe: Composition of Rich User Interface Services“. In: *Proceedings of the 9th International Conference on Web Engineering (ICWE)*. Edition 5648. San Sebastian, Spanien: Springer Berlin/Heidelberg, S. 473–476. ISBN: 978-3-642-02817-5.

Pietschmann, S., Waltsgott, J. und Meißner, K. (Mai 2010c): „A Thin-Server Runtime Platform for Composite Web Applications“. In: *Proceedings of the 5th International Conference on Internet and Web Applications and Services (ICIW)*. Barcelona, Spain: IEEE CPS, S. 390–395. ISBN: 978-0-7695-4022-1.

Tietz, V., Blichmann, G., Pietschmann, S. und Meißner, K. (Juni 2011a): „Task-Based Recommendation of Mashup Components“. In: *Proceedings of the 3rd International Workshop on Lightweight Integration on the Web (ComposableWeb)*. Springer Berlin/Heidelberg.

Tietz, V., Pietschmann, S., Blichmann, G., Meißner, K., Casall, A. und Grams, B. (2011b): „Towards task-based development of enterprise mashups“. In: *Proceedings of the 13th International Conference on Information Integration and Web-based Applications and Services (iiWAS)*. iiWAS '11. New York, NY, USA: ACM, S. 325–328. ISBN: 978-1-4503-0784-0.

Voigt, M., Pietschmann, S. und Grammel, L. (Feb. 2012a): „Context-aware Recommendation of Visualization Components“. In: *Proceedings of the 4th International Conference on Information, Process, and Knowledge Management (eKNOW 2012)*. XPS. ISBN: 978-1-61208-181-6.

Voigt, M., Pietschmann, S. und Meißner, K. (Feb. 2012b): „Towards an End-User-Centered Information Visualization Process for Semantic Web Data“. In: *Proceedings of the 3rd International Workshop on Semantic Models for Adaptive Interacive Systems (SEMAIS)*.

Webreferenzen

Die folgenden Referenzen beziehen sich auf Online-Quellen und sind im Text zur Unterscheidung mit dem Präfix @ gekennzeichnet.

ActiveVOS (2011): *Open Source vs. Open Standards*. Active Endpoints. URL: http://www.activebpel.org/ (besucht am 12. 12. 2011).

Agrawal, A. (25. Juni 2007): *WS-BPEL Extension for People (BPEL4People) Version 1.0*. IBM. URL: http://www.ibm.com/developerworks/webservices/library/specification/ws-bpel4people/ (besucht am 11. 11. 2011).

Akkiraju, R. und Sapkota, B. (Aug. 2007): *Semantic Annotations for WSDL and XML Schema — Usage Guide*. W3C. URL: http://www.w3.org/TR/sawsdl-guide/ (besucht am 11. 11. 2011).

Alves, A. (11. Apr. 2007): *Web Services Business Process Execution Language Version 2.0*. OASIS. URL: http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html (besucht am 11. 11. 2011).

Apache (2011): *Axis2*. Apache Software Foundation. URL: http://axis.apache.org/axis2/java/core/ (besucht am 11. 11. 2011).

ARIS (2011): *MashZone*. Software AG. URL: http://www.mashzone.com/en/mashzone (besucht am 11. 11. 2011).

Beckett, D. und Broekstra, J. (Hrsg.): *SPARQL Query Results XML Format. W3C Recommendation*. W3C. URL: http://www.w3.org/TR/rdf-sparql-XMLres/ (besucht am 01. 02. 2012).

CAS (2011): *PIA*. CAS Software AG. URL: http://www.cas-pia.de/ (besucht am 11. 11. 2011).

Chinnici, R., Moreau, J.-J., Ryman, A. und Weerawarana, S. (26. Juni 2007): *Web Services Description Language (WSDL) Version 2.0*. W3C. URL: http://www.w3.org/TR/wsdl20/ (besucht am 11. 11. 2011).

Clément, L. (17. Aug. 2010): *Web Services Human Task (WS-HumanTask) Version 1.1*. OASIS. URL: http://docs.oasis-open.org/bpel4people/ws-humantask-1.1.html (besucht am 11. 11. 2011).

CometD (2012). The Dojo Foundation. URL: http://cometd.org/ (besucht am 01. 02. 2012).

Corizon (2011): *Enterprise Mashup Platform*. Corizon Ltd. URL: http://www.corizon.com/Products/corizon-enterprise-mashup-platform.html (besucht am 11. 11. 2011).

CRUISe (2012): *Projektwebseite*. CRUISe Projektkonsortium. URL: http://www.cruise-projekt.de/ (besucht am 04. 06. 2012).

Davies, D. (Apr. 2006): *SOA at the user interface*. URL: http://www.looselycoupled.com/opinion/2006/davies-ui-dev0424.html (besucht am 15. 08. 2011).

DEMISA (2011): *Projektwebseite*. Technische Universität Dresden. URL: http://www.mmt.inf.tu-dresden.de/Forschung/Projekte/DEMISA/ (besucht am 11. 11. 2011).

Down, T. (2011): *log4javascript*. URL: http://log4javascript.org/ (besucht am 12. 01. 2012).
Duncan, P. (2007): *PersistJS*. URL: http://pablotron.org/software/persist-js/ (besucht am 12. 01. 2012).
Eclipse (2011a): *Eclipse Modeling Framework Project (EMF)*. The Eclipse Foundation. URL: http://eclipse.org/modeling/emf/ (besucht am 02. 01. 2012).
Eclipse (2011b): *Rich Ajax Platform (RAP)*. Eclipse Foundation. URL: http://www.eclipse.org/rap/ (besucht am 11. 11. 2011).
Eclipse (9. Feb. 2012a): *Equinox*. The Eclipse Foundation. URL: http://www.eclipse.org/equinox/ (besucht am 09. 02. 2012).
Eclipse (2012b): *Model To Text (M2T) Project*. The Eclipse Foundation. URL: http://www.eclipse.org/modeling/m2t/?project=xpand (besucht am 02. 02. 2012).
EDYRA (2011): *Projektwebseite*. Technische Universität Dresden. URL: http://www.mmt.inf.tu-dresden.de/Forschung/Projekte/EDYRA/ (besucht am 11. 11. 2011).
Extensio (2011): *Extender for Microsoft Excel*. Extensio. URL: http://www.extensio.com/products/ExcelExtend.html (besucht am 11. 11. 2011).
Fallside, D. und Walmsley, P. (Hrsg.): *XML Schema Part 0: Primer Secondary Edition*. W3C. URL: http://www.w3.org/TR/xmlschema-0/ (besucht am 11. 11. 2011).
Farrell, J. und Lausen, H. (28. Aug. 2007): *Semantic Annotations for WSDL and XML Schema*. Recommendation. W3C. URL: http://www.w3.org/TR/sawsdl/ (besucht am 11. 11. 2011).
Fielding, R. (Juni 1999): *Hypertext Transfer Protocol – HTTP/1.1, 10 – Status Code Definitions*. RFC 2616. The Internet Society. URL: http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html#sec10 (besucht am 12. 01. 2012).
Google (2011a): *Google Mashup Editor*. URL: http://code.google.com/gme/ (besucht am 11. 11. 2011).
Google (2011b): *Maps API*. Google Inc. URL: http://code.google.com/apis/maps/index.html (besucht am 11. 11. 2011).
Gudgin, M. (Apr. 2007): *SOAP Version 1.2 Part 1: Messaging Framework (Second Edition)*. W3C. URL: http://www.w3.org/TR/soap12-part1/ (besucht am 11. 11. 2011).
Hammer-Lahav, E. (Apr. 2010): *The OAuth 1.0 Protocol*. Internet Engineering Task Force (IETF). URL: http://tools.ietf.org/html/rfc5849 (besucht am 22. 12. 2011).
Heppner, S. (Hrsg.): *JSR 286: Java Portlet Specification Version 2.0*. Java Community Process. URL: http://jcp.org/aboutJava/communityprocess/final/jsr286/index.html (besucht am 11. 11. 2011).
Hickson, I. (8. Dez. 2011): *Web Storage. W3C Candidate Recommendation*. W3C. URL: http://www.w3.org/TR/webstorage/.
Hickson, I. (8. Feb. 2012): *HTML5. A vocabulary and associated APIs for HTML and XHTML*. W3C. URL: http://dev.w3.org/html5/spec/dnd.html#dnd (besucht am 08. 02. 2012).
HyperAdapt (2011): *Projektwebseite*. URL: http://www.hyperadapt.net/ (besucht am 11. 11. 2011).
IBM (2008): *iWidget Specification v1.0*. IBM. URL: http://www-10.lotus.com/ldd/mashupswiki.nsf/dx/iwidget-spec-v1.0.pdf/$file/iwidget-spec-v1.0.pdf.
IBM (2011a): *InfoSphere MashupHub*. IBM. URL: http://www.ibm.com/software/data/infosphere/mashup-hub/ (besucht am 11. 11. 2011).

IBM (2011b): *Mashup Center*. IBM. URL: http://www.ibm.com/software/products/de/de/mashupcenter/ (besucht am 11. 11. 2011).

IBM (2011c): *WebSphere sMash*. IBM. URL: http://www.ibm.com/software/webservers/smash/ (besucht am 11. 11. 2011).

JackBe (2011): *About Presto: The Enterprise App and Mashup Platform*. JackBe Corp. URL: http://www.jackbe.com/enterprise-mashup/content/about-presto (besucht am 11. 11. 2011).

Jena (2011): *A Semantic Web Framework for Java*. URL: http://jena.sourceforge.net/ (besucht am 11. 11. 2011).

jQuery (28. Jan. 2012): *The Write Less, Do More, JavaScript Library*. The jQuery Project. URL: http://jquery.com/ (besucht am 14. 02. 2012).

JUnit.org (3. März 2012): *Resources for Test Driven Development*. URL: http://www.junit.org/ (besucht am 03. 03. 2012).

Kayak (2012): *Labs*. URL: http://www.kayak.com/labs/ (besucht am 20. 02. 2012).

Kesteren, A. (7. Jan. 2012): *XMLHttpRequest Level 2. W3C Working Draft*. W3C. URL: http://www.w3.org/TR/XMLHttpRequest/.

Last.fm (20. Feb. 2012): *Web Services*. URL: http://www.last.fm/api (besucht am 20. 02. 2012).

Mefisto (2011): *Management, Führung, Information, Simulation in Bauwesen - Projektwebseite*. URL: http://www.mefisto-bau.de/ (besucht am 11. 11. 2011).

Niemann, F. (März 2009): *CRM-Anbieter CAS bastelt an ‚Plattform-as-a-Service '*. URL: http://www.computerwoche.de/software/crm/1889151/ (besucht am 11. 11. 2011).

OMA (Sep. 2009): *EMML Documentation*. Open Mashup Alliance. URL: http://www.openmashup.org/omadocs/v1.0/index.html (besucht am 11. 11. 2011).

OMELETTE (2011): *Projektwebseite*. URL: http://www.ict-omelette.eu/ (besucht am 11. 11. 2011).

OMG (Apr. 2006): *Meta Object Facility (MOF) Core Specification 2.0*. Object Management Group (OMG). URL: http://www.omg.org/spec/MOF/2.0 (besucht am 11. 11. 2011).

OMG (1. Feb. 2010): *Object Constraint Language (OCL)*. OMG. URL: http://www.omg.org/spec/OCL/2.2/ (besucht am 02. 02. 2012).

OMG (2011): *OMG Model Driven Architecture*. URL: http://www.omg.org/mda/ (besucht am 31. 05. 2011).

OpenID (5. Dez. 2007): *OpenID Authentication 2.0 - Final*. OpenID Foundation. URL: http://openid.net/specs/openid-authentication-2_0.html.

Oracle (2012): *Project JAXB. JSR 222 Reference Implementation*. Oracle. URL: http://jaxb.java.net/ (besucht am 01. 02. 2012).

OSGi (Mai 2007): *Open Services Gateway Initiative (OSGi) Service Platform Release 4*. Version 4.3. OSGi™Alliance. URL: http://www.osgi.org/Release4/ (besucht am 01. 02. 2012).

Popescu, A. (7. Sep. 2010): *Geolocation API Specification. W3C Candidate Recommendation*. W3C. URL: http://www.w3.org/TR/geolocation-API/ (besucht am 01. 02. 2012).

Prasad, G., Taneja, R. und Todankar, V. (7. Okt. 2007): *Life above the Service Tier. How to Build Application Front-ends in a Service-Oriented World*. (Besucht am 28. 06. 2012).

Qooxdoo (12. Dez. 2011): *Universal JavaScript Framework*. 1&1 Internet AG. URL: http://qooxdoo.org/ (besucht am 02. 02. 2012).

Ruby on Rails (2011). URL: http://rubyonrails.org/ (besucht am 11. 11. 2011).

Russell, A., Wilkins, G., Davis, D. und Nesbitt, M. (2007): *The Bayeux Specification. Bayeux Protocol*. Version 1.0.0. URL: http://svn.cometd.org/trunk/bayeux/bayeux.html (besucht am 01. 02. 2012).

SEKT, P. (Apr. 2005): *PROTON Ontology (PROTo ONtology)*. SEKT Project. URL: http://proton.semanticweb.org/ (besucht am 30. 12. 2011).

Sencha (2012): *Ext Core. Cross-Browser Javascript Library*. Sencha Inc. URL: http://www.sencha.com/products/extcore/ (besucht am 12. 01. 2012).

ServFace (2011): *Project Homepage*. URL: http://www.servface.eu/ (besucht am 11. 11. 2011).

soapUI (2011). SmartBear Software. URL: http://soapui.org (besucht am 22. 03. 2012).

UIMA (2011): *The Apache UIMA Project (Unstructured Information Management applications)*. Apache Softtware Foundation. URL: http://uima.apache.org/ (besucht am 11. 11. 2011).

Vedamuthu, A. S. et al. (Hrsg.): *Web Services Policy 1.5 – Framework*. Recommendation. W3C. URL: http://www.w3.org/TR/ws-policy/ (besucht am 11. 11. 2011).

VVO-Navigator (2012): *Ihr Mobilitätsportal für Dresden und die Region*. Verkehrsverbund Oberelbe GmbH. URL: http://www.vvo-online.de (besucht am 12. 02. 2012).

Waters, K. et al. (Hrsg.): *Delivery Context: Client Interfaces (DCCI) 1.0. Accessing Static and Dynamic Delivery Context Properties*. W3C. URL: http://www.w3.org/TR/DPF/.

WSDL4J (2012): *Web Services Description Language for Java*. URL: http://wsdl4j.sourceforge.net/ (besucht am 01. 02. 2012).

WURFL (14. Feb. 2012): *The Wireless Universal Resource FiLe*. URL: http://wurfl.sourceforge.net/ (besucht am 14. 02. 2012).

Yahoo! (2011a): *Pipes: Rewire the web*. Yahoo! Inc. URL: http://pipes.yahoo.com/pipes/ (besucht am 11. 11. 2011).

Yahoo! (2011b): *YUI Compressor*. Yahoo! Inc. URL: http://developer.yahoo.com/yui/compressor/ (besucht am 12. 01. 2012).

Zalewski, M. (30. März 2011): *Same-Origin Policy*. Google Inc. URL: http://code.google.com/p/browsersec/wiki/Part2#Same-origin_policy (besucht am 11. 11. 2011).

Literaturverzeichnis

Abiteboul, S., Greenshpan, O. und Milo, T. (Okt. 2008): „Modeling the Mashup Space“. In: *Proceedings of the 10th International Workshop on Web Information and Data Management (WIDM)*. Napa Valley, CA, USA: ACM.

Absalom, R. (Aug. 2009): *The Future of Enterprise Mashups. Demand, challenges and vendor opportunities*. Management Report. Business Insights Ltd.

Achilleos, A., Kapitsaki, G. M. und Papadopoulos, G. A. (2011): „A Model-Driven Framework for Developing Web Service Oriented Applications“. In: *Proc. of the 7th International Workshop on Model-Driven Web Engineering at ICWE*.

Alonso, G., Casati, F., Kuno, H. und Machiraju, V. (2004): *Web services: concepts, architectures and applications*. Springer Berlin/Heidelberg. ISBN: 3540440089.

Anderson, C. (2006): *The Long Tail: Why the Future of Business Is Selling Less of More*. New York: Hyperion. ISBN: 1-4013-0237-8.

André, F., Daubert, E. und Gauvrit, G. (2010): „Towards a Generic Context-Aware Framework for Self- Adaptation of Service-Oriented Architectures“. In: *Proceedings of the 5th International Conference on Internet and Web Applications and Services (iiWAS)*, S. 309–314.

Aristoteles (350 B.C.): *Metaphysica*. Bd. H, 1045a.8–10.

Arroyo, S. (Juli 2004): *Data, Information, and Process Integration with Semantic Web Services (DIP): Deliverable D3.1 Report on State of the Art and Requirements analysis*.

Aßmann, U. (2003): *Invasive Software Composition*. New York: Springer Berlin/Heidelberg. ISBN: 978-3540443858.

Baldauf, M., Dustdar, S. und Rosenberg, F. (2007): „A Survey on Context-Aware Systems“. In: *International Journal of Ad Hoc and Ubiquitous Computing* 2.4, S. 263–277.

Baumeister, H. (2005): „Modelling Adaptivity with Aspects“. In: Lowe, D. und Gaedke, M. (Hrsg.): *Web Engineering. Web Engineering*. Lecture Notes in Computer Science 3579, S. 469–486.

Beemer, B. und Gregg, D. (Dez. 2009): „Mashups: A Literature Review and Classification Framework“. In: *Future Internet* 1.1, S. 59–87.

Benslimane, D., Dustdar, S. und Sheth, A. (2008): „Service Mashups“. In: *IEEE Internet Computing* 12.5, S. 13–15.

Bezivin, J. (28. Juni 2011): „On the Future of Software Modeling“. In: *Invited Talk at Technische Universität Dresden*.

Bianchini, D., Antonellis, V. D. und Melchiori, M. (Nov. 2010): „Semantic-driven Mashup Design“. In: *Proceedings of the 12th International Conference on Information Integration and Web-based Applications & Services*, S. 245–252.

Blau, B., Lamparter, S. und Haak, S. (2009): „remash! – Blue- prints for RESTful Situational Web Applications“. In: *Proceedings of the 2nd Workshop on Mashups, Enterprise Mash-ups and Lightweight Composition on the Web*.

Bleul, S., Zapf, M. und Geihs, K. (März 2008): „Self-Integration of Web Services in BPEL Processes“. In: *Proceedings of the SAKS Workshop*. Wiesbaden, Deutschland.

Bowers, S. und Ludäscher, B. (2004): „An Ontology-Driven Framework for Data Transformation in Scientific Workflows“. In: Rahm, E. (Hrsg.): *Proceedings of the 2004 International Workshop on Data Integration in the Life Sciences*. Bd. 2994. Springer Berlin/Heidelberg, S. 1–16.

Bradley, A. (Sep. 2007): *Reference Architecture for Enterprise 'Mashups'*. ID Number: G00151491. Gartner, Inc.

Braga, D., Ceri, S., Martinenghi, D. und Daniel, F. (Sep. 2008): „Mashing Up Search Services“. In: *IEEE Internet Computing* 12, S. 16–23.

Brambilla, M., Comai, S., Fraternali, P. und Matera, M. (2008): „Designing Web Applications with WebML and WebRatio“. In: Rossi, G., Pastor, O., Schwabe, D. und Olsina, L. (Hrsg.): *Web Engineering: Modelling and Implementing Web Applications*. .2. Human-Computer Interaction Series. Springer London, S. 221–261. isbn: 978-1-84628-923-1.

Brodt, A., Nicklas, D., Sathish, S. und Mitschang, B. (2008): „Context-aware mashups for mobile devices“. In: *Proceedings of the International Conference on Web Information Systems Engineering (WISE)*. Bd. 5175. Springer Berlin/Heidelberg, S. 280–291.

Brusilovsky, P. (1996): „Methods and Techniques of Adaptive Hypermedia“. In: *User Modeling and User-Adapted Interaction* 6.2-3, S. 87–129.

Brusilovsky, P. (2001): „Adaptive Hypermedia“. In: *User Modeling and User-Adapted Interaction* 11.1-2, S. 87–110. issn: 0924-1868.

Carlson, M. P., Ngu, A. H., Podorozhny, R. und Zeng, L. (2008): „Automatic Mash Up of Composite Applications“. In: *Proceedings of the 6th International Conference on Service-Oriented Computing*. 5364 Bde. Lecture Notes in Computer Science. Sydney, Australia: Springer Berlin/Heidelberg, S. 317–330. isbn: 978-3-540-89647-0.

Ceri, S., Daniel, F., Matera, M. und Facca, F. (2007): „Model-driven Development of Context-Aware Web Applications“. In: *Transactions on Internet Technology*. ACM.

Chabeb, Y., Tata, S. und Ozanne, A. (Apr. 2010): „YASA-M: A Semantic Web Service Matchmaker“. In: *Proceedings of the International Conference on Advanced Information Networking and Applications*, S. 966–973. issn: 1550-445X.

Chen, H., Perich, F., Finin, T. und Joshi, A. (Aug. 2004): „SOUPA: Standard Ontology for Ubiquitous and Pervasive Applications“. In: *The 1st International Conference on Mobile and Ubiquitous Systems: Networking and Services (MOBIQUITOUS)*, S. 258–267.

Chowdhury, S. R., Rodríguez, C., Daniel, F. und Casati, F. (2011): „Wisdom-Aware Computing: On the Interactive Recommendation of Composition Knowledge“. In: Maximilien, E., Rossi, G., Yuan, S.-T., Ludwig, H. und Fantinato, M. (Hrsg.): *Service-Oriented Computing (ICSOC Workshops)*. Bd. 6568. Lecture Notes in Computer Science. Springer Berlin/Heidelberg, S. 144–155. isbn: 978-3-642-19393-4.

Chudnovsky, O., Gebhardt, H., Weinhold, F. und Gaedke, M. (2011): „Business Process Integration using Telco Mashups“. In: *Proceedings of the 8th International Conference on Mobile Web Information Systems*. Bd. 5. Elsevier, S. 677–680.

Corcho, O., Losada, S. und Benjamins, R. (2007): „Mediation: Bridging between Heterogeneous Web Service Systems“. In: *Semantic Web Services*. Springer Berlin/Heidelberg, S. 287–308. ISBN: 978-3-540-70894-0.

Crane, D. und McCarthy, P. (2008): *Comet and Reverse Ajax: The Next Generation Ajax 2.0*. Apress. ISBN: 978-1590599983.

Curbera, F., Duftler, M., Khalaf, R. und Lovell, D. (2007): „Bite: Workflow Composition for the Web“. In: *Proceedings of the International Conference on Service Oriented Computing (ICSOC 2007)*. Bd. 4749. Springer Berlin/Heidelberg, S. 94–106.

Daniel, F. (Feb. 2010): *Context-Aware Web Applications - The Model-Driven Way*. Bd. 1. VDM Verlag. ISBN: 978-3639113938.

Daniel, F., Casati, F., Benatallah, B. und Shan, M.-C. (Nov. 2009): „Hosted Universal Composition: Models, Languages and Infrastructure in mashArt“. In: *Proceedings of the 28th International Conference on Conceptual Modeling*.

Daniel, F. und Matera, M. (Sep. 2008): „Mashing Up Context-Aware Web Applications: A Component-Based Development Approach“. In: *Proceedings of the 10th International Conference on Web Information Systems Engineering (WISE)*. Bd. 5175. Lecture Notes in Computer Science. Auckland, New Zealand: Springer Berlin/Heidelberg, S. 250–263.

Daniel, F. und Matera, M. (2009): „Turning Web Applications into Mashup Components: Issues, Models, and Solutions“. In: *Proceedings of the 9th International Conference on Web Engineering*. ICWE 2009. San Sebastian, Spain: Springer Berlin/Heidelberg, S. 45–60. ISBN: 978-3-642-02817-5.

Dannecker, L., Feldmann, M., Nestler, T., Hübsch, G., Jugel, U. und Muthmann, K. (2010): „Rapid Development of Composite Applications Using Annotated Web Services“. In: DANIEL, F. und FACCA, F. (Hrsg.): *Current Trends in Web Engineering*. Bd. 6385. Lecture Notes in Computer Science. Springer Berlin/Heidelberg, S. 1–12. ISBN: 978-3-642-16984-7.

Deursen, A., Klint, P. und Visser, J. (Juni 2000): „Domain-Specific Languages: An Annotated Bibliography“. In: *ACM SIGPLAN Notices* 35.6 (6), S. 26–36. ISSN: 0362-1340.

Dey, A. K. und Abowd, G. D. (1999): „Towards a better understanding of context and context-awareness“. In: *CHI 2000 Workshop on the What, Who, Where, When, and How of Context-Awareness*. Bd. 4. Springer Berlin/Heidelberg, S. 1–6.

Di Lorenzo, G., Hacid, H., Paik, H. und Benatallah, B. (2009): „Data Integration in Mashups“. In: *ACM SIGMOD Record* 38.1, S. 59–66. ISSN: 0163-5808.

Dustdar, S. und Schreiner, W. (2005): „A Survey on Web Services Composition“. In: *International Journal of Web and Grid Services* 1.1, S. 1–30.

Ennals, R. und Trushkowsky, B. (2008): „Participatory Mashups: Using Users to make Data Mashable“. In: *Workshop on Tinkering Tailoring and Mashing at CSCW*.

Erl, T. (2005): *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall.

Erradi, A., Maheshwari, P. und Tosic, V. (2006): „Policy-Driven Middleware for Self-adaptation of Web Services Compositions“. In: *Proceedings of the International Conference on Middleware*, S. 62–80.

Estublier, J., Dieng, I. A. und Simon, E. (2010): „Automating Component Selection and Building Flexible Composites for Service-Based Applications“. In: MACIASZEK, L. A., GONZÁLEZ-PÉREZ, C. und JABLONSKI, S. (Hrsg.): *Evaluation of Novel Approaches*

to Software Engineering. Bd. 69. Communications in Computer and Information Science. Springer Berlin/Heidelberg, S. 93–106. ISBN: 978-3-642-14819-4.

Eugster, P., Felber, P., Guerraoui, R. und Kermarrec, A.-M. (Juni 2003): „The Many Faces of Publish/Subscribe". In: *ACM Computing Surveys (CSUR)* 35 (2), S. 114–131. ISSN: 0360-0300.

Euzanat, J. und Shvaiko, P. (2007): *Ontology Matching*. Bd. 1. Springer Berlin/Heidelberg. 334 S. ISBN: 978-3540496113.

Faison, T. (2006): *Event-based programming: taking events to the limit*. Apress. ISBN: 1-59059-643-9.

Feldmann, M., Nestler, T., Jugel, U., Muthmann, K., Hübsch, G. und Schill, A. (2009): „Overview of an End-user enabled Model-driven Development Approach for Interactive Applications based on Annotated Services". In: *Proceedings of the 4th Workshop on Emerging Web Services Technology*. ACM.

Fensel, D., Kerrigan, M. und Zaremba, M. (2008): *Implementing Semantic Web Services: The SESA Framework*. Springer Berlin/Heidelberg.

Fernández, R., Lizcano, D., Ortega, S. und Soriano, J. (2009): „Towards a user-centered composition system for service-based composite applications". In: *Proceedings of the 11th International Conference on Information Integration and Web-based Applications & Services*. ACM, S. 321–330. ISBN: 978-1-60558-660-1.

Fischer, T., Bakalov, F., König-Ries, B., Nauerz, A. und Welsch, M. (2009): „An Evolutionary Algorithm for Automatic Composition of Information-gathering Web Services in Mashups". In: *Proceedings of the 7th IEEE European Conference on Web Services*. IEEE, S. 39–48. ISBN: 978-0-7695-3854-9.

Fluegge, M., Tizzo, N. P., Santos, I. J. G. D. und Madeira, E. R. M. (2006): „Challenges and Techniques on the Road to Dynamically Compose Web Services". In: *Proceedings of the 6th International Conference on Web Engineering*. ACM, S. 40–47.

Fraternali, P., Rossi, G. und Sá andnchez-Figueroa, F. (Mai 2010): „Rich Internet Applications". In: *IEEE Internet Computing* 14.3, S. 9–12. ISSN: 1089-7801.

Fujii, K. und Suda, T. (2004): „Dynamic Service Composition Using Semantic Information". In: *Proceedings of the 2nd International Conference on Service Oriented Computing*. ACM, S. 39–48. ISBN: 1-58113-871-7.

Fujii, K. und Suda, T. (Mai 2009): „Semantics-based context-aware dynamic service composition". In: *ACM Transactions on Autonomous and Adaptive Systems* 4 (2), 12:1–12:31. ISSN: 1556-4665.

Gaedke, M., Beigl, M., Gellersen, H.-W. und Segor, C. (1999a): „Web Content Delivery to Heterogeneous Mobile Platforms". In: *Advances in Database Technologies: ER'98 Workshops on Data Warehousing and Data Mining, Mobile Data Access, and Collaborative Work Support and Spatio-Temporal Data Management*. Springer Verlag, S. 205.

Gaedke, M., Rehse, J. und Graef, G. (1999b): „A Repository to facilitate Reuse in Component-Based Web Engineering". In: *International Workshop on Web Engineering*. Toronto, Canada.

Gamma, E., Helm, R., Johnson, R. und Vlissides, J. (März 1995): *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley. ISBN: 0201633612.

Garlan, D., Schmerl, B. und Cheng, S.-W. (2009): „Software Architecture-Based Self-Adaptation“. In: ZHANG, Y., YANG, L. T. und DENKO, M. K. (Hrsg.): *Autonomic Computing and Networking*. Springer Berlin/Heidelberg, S. 31–55. ISBN: 978-0-387-89828-5.

Garrigós, I., Meliá, S. und Casteleyn, S. (2009): „Adapting the Presentation Layer in Rich Internet Applications“. In: *Proceedings of the 9th International Conference on Web Engineering*. Bd. 5648. Lecture Notes in Computer Science. Springer Berlin/Heidelberg, S. 292.

Gartner (Okt. 2007): *Gartner Identifies the Top 10 Strategic Technologies for 2008*. http://www.gartner.com/it/page.jsp?id=530109. Gartner Inc.

Garzotto, F., Paolini, P. und Schwabe, D. (Jan. 1993): „HDM – A Model-Based Approach to Hypertext Application Design“. In: *ACM Transactions on Information Systems (TOIS)* 11 (1), S. 1–26. ISSN: 1046-8188.

Gellersen, H.-W. und Gaedke, M. (Jan. 1999): „Object-Oriented Web Application Development“. In: *IEEE Internet Computing* 3.1, S. 60–68. ISSN: 1089-7801.

Gilles, F., Hoyer, V., Janner, T. und Stanoevska-Slabeva, K. (2009): „Lightweight composition of ad-hoc enterprise-class applications with context-aware enterprise mashups“. In: *Proceedings of the International Conference on Service-Oriented Computing*. Stockholm, Sweden: Springer Berlin/Heidelberg, S. 509–519. ISBN: 3-642-16131-6, 978-3-642-16131-5.

Gjørven, E., Eliassen, F. und Rouvoy, R. (2008a): „Experiences from Developing a Component Technology Agnostic Adaptation Framework“. In: *Proceedings of the 11th International Symposium on Component-Based Software Engineering*. Karlsruhe, Germany: Springer Berlin/Heidelberg, S. 230–245. ISBN: 978-3-540-87890-2.

Gjørven, E., Rouvoy, R. und Eliassen, F. (2008b): „Cross-layer Self-adaptation of Service-oriented Architectures“. In: *Proceedings of the 3rd Workshop on Middleware for Service Oriented Computing*. Leuven, Belgium: ACM, S. 37–42. ISBN: 978-1-60558-368-6.

Gómez, J., Cachero, C. und Pastor, O. (2001): „On Conceptual Modeling of Device-Independent Web Applications: Towards a Web-Engineering Approach“. In: *IEEE Multimedia* 8.2, S. 20–32.

Graef, G. und Gaedke, M. (2000): „Construction of Adaptive Web-Applications from Reusable Components“. In: BAUKNECHT, K., MADRIA, S. und PERNUL, G. (Hrsg.). *Electronic Commerce and Web Technologies*. Lecture Notes in Computer Science 1875, S. 1–13.

Grammel, L. und Storey, M.-A. (2010): „A Survey of Mashup Development Environments“. In: CHIGNELL, M., CORDY, J., NG, J. und YESHA, Y. (Hrsg.): *The Smart Internet*. Bd. 6400. Lecture Notes in Computer Science. Springer Berlin/Heidelberg, S. 137–151.

Gui, N., Florio, V., Sun, H. und Blondia, C. (2009): „ACCADA: A Framework for Continuous Context-Aware Deployment and Adaptation“. In: *Proceedings of the 11th International Symposium on Stabilization, Safety, and Security of Distributed Systems*. Lyon, France, S. 325–340. ISBN: 978-3-642-05117-3.

Guo, R., Zhu, B. B., Feng, M., Pan, A. und Zhous, B. (2008): „CompoWeb: A Component-Oriented Web Architecture“. In: *Proceeding of the 17th International*

Conference on World Wide Web. Beijing, China: ACM, S. 545–554. ISBN: 978-1-60558-085-2.

Hinz, M. (Feb. 2008): „Kontextsensitive Generierung adaptiver multimedialer Webanwendungen“. Dissertation. Technische Universität Dresden.

Hinz, M., Pietschmann, S. und Fiala, Z. (Juni 2007): „A Framework for Context Modeling in Adaptive Web Applications“. In: *IADIS International Journal of WWW/Internet* 5.1.

Hohpe, G. und Woolf, B. (2003): *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. 1. Aufl. Addison-Wesley. ISBN: 978-0321200686.

Hoyer, V. und Fischer, M. (2008): „Market Overview of Enterprise Mashup Tools“. In: *Proceedings of the 6th International Conference on Service Oriented Computing (ICSOC)*. Bd. 5364. Springer Berlin/Heidelberg, S. 708–721.

Hristoskova, A., Volckaert, B. und Turck, F. D. (2011): „Framework Managing the Automated Construction and Runtime Adaptation of Service Mashups“. In: *Proceedings of the 2nd International Workshop on Semantic Interoperability*, S. 31.

Hristoskova, A., Volckaert, B., Turck, F. D. und Dhoedt, B. (2010): „Design of a Framework for Automated Service Mashup Creation and Execution Based on Semantic Reasoning“. In: *Proceedings of the 5th International Conference on Internet and Web Applications and Services*. IEEE CPS, S. 149–154.

Hübsch, G., Liebing, C., Spillner, J. und Schill, A. (2010): „A Description Language for User Interface Services“. In: *Proceedings of the IADIS International Conference on WWW/Internet*. Timisoara, Romania. ISBN: 978-972-8939-25-0.

Irmert, F., Fischer, T. und Meyer-Wegener, K. (2008): „Runtime Adaptation in a Service-Oriented Component Model“. In: *SEAMS'08: Proceedings of the International Workshop on Software Engineering for Adaptive and Self-Managing Systems*. Leipzig, Germany: ACM, S. 97–104. ISBN: 978-1-60558-037-1.

Jaeger, M. C., Rojec-Goldmann, G., Liebetruth, C., Mühl, G. und Geihs, K. (Feb. 2005): „Ranked Matching for Service Descriptions Using OWL-S“. In: *Kommunikation in verteilten Systemen (KiVS 2005)*. Informatik Aktuell. Springer Press, S. 91–102. ISBN: 3-540-24473-5.

Jhingran, A. (2006): „Enterprise Information Mashups: Integrating Information, Simply“. In: *Proceedings of the 32nd International Conference on Very Large Data Bases (VLDB)*. VLDB Endowment. ACM, S. 3–4.

Kapitsaki, G. M., Kateros, D. A., Pappas, C. A., Tselikas, N. D. und Venieris, I. S. (2008): „Model-Driven Development of Composite Web Applications“. In: *Proceedings of the 10th International Conference on Information Integration and Web-based Applications & Services (iiWAS)*. ACM, S. 399–402.

Karol, S., Niederhausen, M., Kadner, D., Aßmann, U. und Meißner, K. (Sep. 2011): „Detecting and Resolving Conflicts between Adaptation Aspects in Multi-staged XML Transformations“. In: *Proceedings of the 11th ACM Symposium on Document Engineering (DocEng'11)*. ACM, S. 229–238. ISBN: 978-1-4503-08.

Kateros, D. A., Kapitsaki, G. M., Tselikas, N. D. und Venieris, I. S. (2008): „A Methodology for Model-Driven Web Application Composition“. In: *Proceedings of the 2008 IEEE International Conference on Services Computing*. Bd. 2. IEEE, S. 489–492. ISBN: 978-0-7695-3283-7-02.

Ketfi, A., Belkhatir, N. und Cunin, P.-Y. (2002): „Automatic Adaptation of Component-based Software: Issues and Experiences“. In: *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, S. 1365–1371.

Ketter, W., Banjanin, M., Guikers, R. und Kayser, A. (2009): „Introducing an Agile Method for Enterprise Mash-Up Component Development“. In: *Proceedings of the International IEEE Conference on Commerce and Enterprise Computing*. IEEE, S. 293–300. ISBN: 978-0-7695-3755-9.

Kiczales, G. (Juni 1997): „Aspect-Oriented Programming“. In: AKSIT, M. und MATSUOKA, S. (Hrsg.): *Proceedings of the 11th European Conference on Object-Oriented Programming*. Bd. 1241. Lecture Notes in Computer Science. Springer Berlin/Heidelberg, S. 220–242. ISBN: 978-3-540-63089-0.

Kim, Y. und Stohr, E. A. (1998): „Software Reuse: Survey and Research Directions“. In: *Journal of Management Information Systems* 14.4, S. 113–147.

King, A. B. (2003): *Speed Up Your Site: Web Site Optimization*. New Riders. ISBN: 978-0735713246.

Kleshchev, A. und Gribovy, V. (2003): „From an Ontology-Oriented Approach Conception to User Interface Development“. In: *Information Theories & Applications* 10.1, S. 87–94.

Klusch, M. (2008): „Semantic Web Service Coordination“. In: *CASCOM: Intelligent Service Coordination in the Semantic Web*. Software Agent Technologies and Autonomic Computing. Kap. 4. ISBN: 978-3-7643-8575-0.

Klusch, M., Kapahnke, P. und Zinnikus, I. (Juli 2009): „SAWSDL-MX2: A Machine-Learning Approach for Integrating Semantic Web Service Matchmaking Variants“. In: *Proceedings of the 7th International Conference on Web Services*, S. 335–342.

Knapp, A., Koch, N., Wirsing, M. und Zhang, G. (2007): „UWE - Ein Ansatz zur modellgetriebenen Entwicklung von Webanwendungen“. In: *i-com* 6.3/2007, S. 5–12.

Kopecký, J. und Vitvar, T. (Aug. 2008): „WSMO-Lite: Lowering the Semantic Web Services Barrier with Modular and Light-Weight Annotations“. In: *Proceedings of the International Conference on Semantic Computing*. IEEE, S. 238–244. ISBN: 978-0-7695-3279-0.

Koschmider, A., Torres, V. und Pelechano, V. (Apr. 2009): „Elucidating the Mashup Hype: Definition, Challenges, Methodical Guide and Tools for Mashups“. In: *Proceedings of the 2nd Workshop on Mashups, Enterprise Mashups and Lightweight Composition on the Web*.

Kreska, T. S. (Nov. 2008): „Entwicklung eines generischen Service-Wrappers für dienstbasierte Benutzerschnittstellen“. Diplomarbeit. Lehrstuhl für Multimediatechnik. Technische Universität Dresden.

Kuhn, H. W. (1955): „The Hungarian Method for the Assignment Problem“. In: *Naval research logistics quarterly* 2.1-2, S. 83–97. ISSN: 1931-9193.

Küster, U. und König-Ries, B. (2007): „Dynamic Binding for BPEL Processes – A Lightweight Approach to Integrate Semantics into Web Services“. In: GEORGAKOPOULOS, D. et al. (Hrsg.): *Proceedings of the 2nd International Workshop on Engineering Service-Oriented Applications at ICSOC06*. Springer Berlin/Heidelberg, S. 116–127. ISBN: 978-3-540-75491-6.

Laga, N., Bertin, E. und Crespi, N. (2009): „A Web Based Framework for Rapid Integration of Enterprise Applications“. In: *Proceedings of the 2009 International Conference on Pervasive services*. ACM, S. 189–198.

Li, L. und Horrocks, I. (Mai 2003): „A Software Framework for Matchmaking Based on Semantic Web Technology“. In: *Proceedings of the 12th International Conference on World Wide Web*. Budapest, Hungary: ACM, S. 331–339. ISBN: 1-58113-680-3.

Li, Y., Fang, J. und Xiong, J. (2008): „A Context-Aware Services Mash-Up System“. In: *Proceedings of the 7th International Conference on Grid and Cooperative Computing*. Washington, DC, USA: IEEE Computer Society, S. 707–712. ISBN: 978-0-7695-3449-7.

Linaje, M., Preciado, J. C. und Sánchez-Figueroa, F. (Nov. 2007): „Engineering Rich Internet Application User Interfaces over Legacy Web Models“. In: *IEEE Internet Computing* 11.6, S. 53–59. ISSN: 1089-7801.

Liu, X., Hui, Y., Sun, W. und Liang, H. (2007): „Towards Service Composition Based on Mashup“. In: *IEEE Congress on Services*, S. 332–339.

Lizcano, D., Soriano, J., Reyes, M. und Hierro, J. J. (2008): „EzWeb/FAST: Reporting on a successful mashup-based solution for developing and deploying composite applications in the upcoming web of services“. In: *Proceedings of the 10th International Conference on Information Integration and Web-based Applications & Services*. New York, NY, USA: ACM, S. 15–24.

López, J., Bellas, F., Pan, A. und Montoto, P. (2009): „A Component-Based Approach for Engineering Enterprise Mashups“. In: *Proceedings of the 9th International Conference on Web Engineering*. Bd. 5648. Lecture Notes in Computer Science. Springer Berlin/Heidelberg, S. 30–44. ISBN: 978-3-642-02817-5.

López, J., Pan, A., Bellas, F. und Montoto, P. (2008): „Towards a Reference Architecture for Enterprise Mashups“. In: *Actas de los Talleres de las Jornadas de IngenierÌa del Software y Bases de Datos* 2.2, S. 67–76.

Manolescu, I., Brambilla, M., Ceri, S., Comai, S. und Fraternali, P. (Aug. 2005): „Model-Driven Design and Deployment of Service-Enabled Web Applications“. In: *ACM Transactions on Internet Technology (TOIT)* 5.3, S. 439–479.

Marconi, A. und Pistore, M. (2009): „Synthesis and Composition of Web Services“. In: Bernardo, M., Padovani, L. und Zavattaro, G. (Hrsg.): *Formal Methods for Web Services*. Bd. 5569. Lecture Notes in Computer Science. Springer Berlin/Heidelberg, S. 89–157.

Martin, D., Paolucci, M. und Wagner, M. (Nov. 2007a): „Bringing semantic annotations to web services: OWL-S from the SAWSDL perspective“. In: *Proceedings of the 6th International Semantic Web Conference*. Busan, Korea: Springer Berlin/Heidelberg, S. 340–352. ISBN: 3-540-76297-3, 978-3-540-76297-3.

Martin, D. (Sep. 2007b): „Bringing Semantics to Web Services with OWL-S“. In: *World Wide Web* 10 (3), S. 243–277. ISSN: 1386-145X.

Maximilien, E. M., Ranabahu, A. und Gomadam, K. (Sep. 2008): „An Online Platform for Web APIs and Service Mashups“. In: *IEEE Internet Computing* 12.5, S. 32–43. ISSN: 1089-7801.

Maximilien, E. M., Ranabahu, A. und Tai, S. (2007): „Swashup: Situational Web Applications Mashups“. In: *Companion to the 22nd Conference on Object-Oriented Programming Systems and Applications (OOPSLA)*. New York, NY, USA: ACM, S. 797–798. ISBN: 978-1-59593-865-7.

Meliá, S., Gómez, J., Pérez, S. und Díaz, O. (2008): „A Model-Driven Development for GWT-Based Rich Internet Applications with OOH4RIA“. In: *ICWE '08: Proceedings of the 2008 Eighth International Conference on Web Engineering*. Washington, DC, USA: IEEE Computer Society, S. 13–23. ISBN: 978-0-7695-3261-5.

Mikkonen, T. und Taivalsaari, A. (Nov. 2010): „The Mashware Challenge: Bridging the Gap Between Web Development and Software Engineering“. In: *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*. ACM, S. 245–250. ISBN: 978-1-4503-0427-6.

Mitschick, A. (März 2010): „Ontologiebasierte Indexierung und Kontextualisierung multimedialer Dokumente für das persönliche Wissensmanagement“. Technische Universität Dresden.

Mitschick, A., Pietschmann, S. und Meißner, K. (Feb. 2010): „An Ontology-Based, Cross-Application Context Modeling and Management Service“. In: *International Journal on Semantic Web and Information Systems (IJSWIS)*.

Mukhija, A. und Glinz, M. (2005): „Runtime Adaptation of Applications through Dynamic Recomposition of Components“. In: *Systems Aspects in Organic and Pervasive Computing - ARCS 2005*. Lecture Notes in Computer Science. Springer Verlag, S. 124–138. ISBN: 3-540-25273-8.

Myers, B. A. und Rosson, M. B. (1992): „Survey on User Interface Programming“. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. Monterey, California, United States: ACM, S. 195–202. ISBN: 0-89791-513-5.

Nagarajan, M., Verma, K., Sheth, A. P. und Miller, J. A. (2007): „Ontology Driven Data Mediation in Web Services“. In: *International Journal of Web Services Research* 4.4, S. 104–126.

Nestler, T., Feldmann, M., Preußner, A. und Schill, A. (Juni 2009): „Service Composition at the Presentation Layer using Web Service Annotations“. In: *Proceedings of the 1st International Workshop on Lightweight Integration on the Web (ComposableWeb)*.

Ngu, A., Carlson, M., Sheng, Q. und Paik, H. (Jan. 2010): „Semantic-Based Mashup of Composite Applications“. In: *IEEE Transactions on Services Computing* 3.1.

Niederhausen, M., Pietschmann, S., Ruch, T. und Meißner, K. (Apr. 2010). "Web-Based Support By Thin-Client Co-Browsing". In: BADR, Y., CHBEIR, R., ABRAHAM, A. und HASSANIEN, A.-E. (Hrsg.): *Emergent Web Intelligence: Advanced Semantic Technologies* Bd. XVI. 544. Springer Berlin/Heidelberg. ISBN: 978-1-84996-076-2.

Nilsson, E. G., Floch, J., Hallsteinsen, S. und Stav, E. (2006): „Model-based user interface adaptation“. In: *Computers & Graphics* 30.5, S. 692–701.

Noy, N. F. (2009): „Ontology Mapping“. In: STAAB, S. und STUDER, R. (Hrsg.): *Handbook on Ontologies*. International Handbooks on Information Systems. Springer Berlin/Heidelberg, S. 573–590. ISBN: 978-3-540-92673-3.

O'Sullivan, J. (2006): „Towards a Precise Understanding of Service Properties“. Queensland University of Technology.

Papazoglou, M. P. und Yang, J. (2002): „Design Methodology for Web Services and Business Processes“. In: BUCHMANN, A., FIEGE, L., CASATI, F., HSU, M.-C. und SHAN, M.-C. (Hrsg.): *Technologies for E-Services*. Bd. 2444. Lecture Notes in Computer Science. Springer Berlin/Heidelberg, S. 175–233.

Paternò, F., Santoro, C. und Spano, L. D. (Nov. 2009): „MARIA: A Universal, Declarative, Multiple Abstraction-Level Language for Service-Oriented Applications

in Ubiquitous Environments“. In: *ACM Transactions on Computer-Human Interaction* 16.4, S. 19.

Paulheim, H. (2009): „Ontology-based modularization of user interfaces“. In: *Proceedings of the 1st Symposium on Engineering Interactive Computing Ssystems*. ACM, S. 23–28.

Paulheim, H. und Probst, F. (2010): „Application integration on the user interface level: An ontology-based approach“. In: *Data & Knowledge Engineering* 69.11, S. 1103–1116.

Pautasso, C. (2009): „Composing RESTful Services with JOpera“. In: *Proceedings of the 8th International Conference on Software Composition*. Bd. 5634. Lecture Notes in Computer Science. Springer Berlin/Heidelberg, S. 142–159.

Pautasso, C. und Alonso, G. (Apr. 2005): „Flexible Binding for Reusable Composition of Web Services“. In: *Proceedings of the 4th Workshop on Software Composition*. Springer Berlin/Heidelberg, S. 151–166.

Penta, M. D., Esposito, R., Villani, M. L., Codato, R., Colombo, M. und Nitto, E. D. (2006): „WS Binder: A Framework to Enable Dynamic Binding of Composite Web Services“. In: *Proceedings of the 2006 International Workshop on Service-Oriented Software Engineering*. ACM. ACM, S. 74–80. ISBN: 1-59593-398-0.

Pietschmann, S. (2009): „A Model-Driven Development Process and Runtime Platform for Adaptive Composite Web Applications“. In: *International Journal on Advances in Internet Technology* 2.4, S. 277–288. ISSN: 1942-2652.

Pietschmann, S., Mitschick, A., Winkler, R. und Meißner, K. (Dez. 2008): „CroCo: Ontology-Based, Cross-Application Context Management“. In: *Proceedings of the 3rd International Workshop on Semantic Media Adaptation and Personalization*. Prag, Tschechische Republik: IEEE CPS. ISBN: 978-0-7695-3444-2.

Pietschmann, S., Radeck, C. und Meißner, K. (Sep. 2011b): „Semantics-Based Discovery, Selection and Mediation for Presentation-Oriented Mashups“. In: *Proceedings of the 5th International Workshop on Web APIs and Service Mashups*. New York, NY, USA: ACM. ISBN: 978-1-4503-0823-6.

Pietschmann, S., Tietz, V., Reimann, J., Liebing, C., Pohle, M. und Meißner, K. (Nov. 2010a): „A Metamodel for Context-Aware Component-Based Mashup Applications“. In: *Proceedings of the 12th International Conference on Information Integration and Web-based Applications & Services (iiWAS)*. ACM. ISBN: 978-1-4503-0421-4.

Pietschmann, S., Voigt, M. und Meißner, K. (Okt. 2009a): „Adaptive Rich User Interfaces for Human Interaction in Business Processes“. In: *Proceedings of the 10th International Conference on Web Information Systems Engineering (WISE)*. WISE. Posen, Polen: Springer Berlin/Heidelberg. ISBN: 978-3-642-04408-3.

Pietschmann, S., Voigt, M. und Meißner, K. (Mai 2009b): „Dynamic Composition of Service-Oriented Web User Interfaces“. In: *Proceedings of the 4th International Conference on Internet and Web Applications and Services (ICIW)*. Mestre/Venedig, Italien: IEEE CPS, S. 217–222. ISBN: 978-0-7695-3613-2.

Pietschmann, S., Voigt, M., Rümpel, A. und Meißner, K. (Juni 2009c): „CRUISe: Composition of Rich User Interface Services“. In: *Proceedings of the 9th International Conference on Web Engineering (ICWE)*. Edition 5648. San Sebastian, Spanien: Springer Berlin/Heidelberg, S. 473–476. ISBN: 978-3-642-02817-5.

Pietschmann, S., Waltsgott, J. und Meißner, K. (Mai 2010b): „A Thin-Server Runtime Platform for Composite Web Applications“. In: *Proceedings of the 5th International*

Conference on Internet and Web Applications and Services (ICIW). Barcelona, Spain: IEEE CPS, S. 390–395. ISBN: 978-0-7695-4022-1.

Preciado, J. C., Linaje, M., Comai, S. und Sánchez-Figueroa, F. (Okt. 2007): „Designing Rich Internet Applications with Web Engineering Methodologies“. In: *Proceedings of the 9th International Workshop on Web Site Evolution*, S. 23–30.

Preciado, J. C. (2008): „Designing Rich Internet Applications Combining UWE and RUX-Method“. In: *Proceedings of the 8th International Conference on Web Engineering*, S. 148–154. ISBN: 978-0-7695-3261-5.

Radeck, C. (Mai 2010): „Kontextmodellierungs- und Adaptionsmechanismen für komposite Webanwendungen“. Großer Beleg. Lehrstuhl für Multimediatechnik. Technische Universität Dresden.

Radeck, C. (Feb. 2011): „Unterstützung semantischer Komposition in Mashups“. Diplomarbeit. Lehrstuhl für Multimediatechnik. Technische Universität Dresden.

Rao, J. und Su, X. (2005): „A Survey of Automated Web Service Composition Methods“. In: CARDOSO, J. und SHETH, A. (Hrsg.): *Semantic Web Services and Web Process Composition*. Bd. 3387. Lecture Notes in Computer Science. Springer Berlin/Heidelberg, S. 43–54.

Reimann, J. (Okt. 2009): „Generisches Kompositionsmodell für UI-Mashups“. Großer Beleg. Lehrstuhl für Multimediatechnik. Technische Universität Dresden.

Rümpel, A., Baumgärtel, K. und Meißner, K. (Nov. 2010): „Platform Adaptation of Mashup UI Components“. In: *Proceedings of the 2nd International Conference on Adaptive and Self-adaptive Systems and Applications (ADAPTIVE)*, S. 164–169. ISBN: 978-1-61208-0.

Schmidt, K.-U., Stühmer, R. und Stojanovic, L. (2009): „Gaining Reactivity for Rich Internet Applications by Introducing Client-side Complex Event Processing and Declarative Rules“. In: *Proceedings of the AAAI Spring Symposium on Intelligent Event Processing*, S. 67–72.

Shaer, O. und Hornecker, E. (2010): *Tangible User Interfaces*. Now Publishers. ISBN: 978-1601983282.

Sheth, A. P., Gomadam, K. und Lathem, J. (Nov. 2007): „SA-REST: Semantically Interoperable and Easier-to-Use Services and Mashups“. In: *IEEE Internet Computing* 11, S. 91–94. ISSN: 1089-7801.

Shvaiko, P. und Euzenat, J. (2005): „A Survey of Schema-Based Matching Approaches“. In: SPACCAPIETRA, S. (Hrsg.): *Journal on Data Semantics IV*. Bd. 3730. Lecture Notes in Computer Science. Springer Berlin/Heidelberg, S. 146–171.

Simmen, D. E., Altinel, M., Markl, V., Padmanabhan, S. und Singh, A. (2008): „Damia: Data Mashups for Intranet Applications“. In: *Proceedings of the 2008 ACM SIGMOD International Conference on Management Of Data*. SIGMOD '08. Vancouver, Canada: ACM, S. 1171–1182. ISBN: 978-1-60558-102-6.

Sivashanmugam, K., Verma, K., Sheth, A. und Miller, J. (2003): „Adding Semantics to Web Services Standards“. In: *Proceedings of the International Conference on Web Services*. Citeseer, S. 395–401.

Stahl, T., Völter, M., Efftinge, S. und Haase, A. (2007): *Modellgetriebene Softwareentwicklung. Techniken, Engineering, Management*. 2. Aufl. dPunkt. ISBN: 978-3-89864-448-8.

Steger, M. und Kappert, C. (März 2008): „User-facing SOA“. In: *Java Magazin*, S. 65–77.

Szomszor, M., Payne, T. R. und Moreau, L. (Sep. 2006): „Automated Syntactic Medation for Web Service Integration“. In: *Proceedings of the IEEE International Conference on Web Services*, S. 127–136. ISBN: 0-7695-2669-1.

Szyperski, C. (Nov. 2002): *Component Software: Beyond Object-Oriented Programming*. 2. Aufl. Addison-Wesley. ISBN: 0-201-74572-0.

Tietz, V., Blichmann, G., Pietschmann, S. und Meißner, K. (Juni 2011): „Task-Based Recommendation of Mashup Components“. In: *Proceedings of the 3rd International Workshop on Lightweight Integration on the Web (ComposableWeb)*. Springer Berlin/Heidelberg.

Tilsner, M., Fiech, A. und Specht, T. (Mai 2009): „Integrating Heterogeneous User Interfaces in Service Oriented Web Applications“. In: *Proceedings of the 2nd Canadian Conference on Computer Science and Software Engineering*. ACM, S. 73–81. ISBN: 978-1-60558-401-0.

Treiber, M., Kritikos, K., Schall, D., Dustdar, S. und Plexousakis, D. (2010): „Modeling context-aware and socially-enriched mashups“. In: *Proceedings of the 3rd and 4th International Workshop on Web APIs and Services Mashups*. Ayia Napa, Cyprus: ACM, 2:1–2:8. ISBN: 978-1-4503-0418-4.

Tsai, W.-T., Huang, Q., Elston, J. und Chen, Y. (2008): „Service-Oriented User Interface Modeling and Composition“. In: *Proceedings of the 2008 International Conference on e-Business Engineering*. IEEE, S. 21–28.

Tuchinda, R., Szekely, P. und Knoblock, C. A. (2008): „Building Mashups by Example“. In: *Proceedings of the 13th International Conference on Intelligent User Interfaces*. Gran Canaria, Spain: ACM, S. 139–148. ISBN: 978-1-59593-987-6.

Van Acker, S., De Ryck, P., Desmet, L., Piessens, F. und Joosen, W. (Dez. 2011): „WebJail: Least-privilege integration of third-party components in web mashups“. In: *Proceedings of the 27th Annual Computer Security Applications Conference*. Bd. 1. 1, S. 307–316. ISBN: 978-1-4503-0672-0.

Voigt, M. (Apr. 2009): „Dienstbasierte Benutzerschnittstellen zur menschlichen Interaktion in Geschäftsprozessen“. Diplomarbeit. Lehrstuhl für Multimediatechnik. Technische Universität Dresden.

Voigt, M., Pietschmann, S. und Grammel, L. (Feb. 2012a): „Context-aware Recommendation of Visualization Components“. In: *Proceedings of the 4th International Conference on Information, Process, and Knowledge Management (eKNOW 2012)*. XPS. ISBN: 978-1-61208-181-6.

Voigt, M., Pietschmann, S. und Meißner, K. (Feb. 2012b): „Towards an End-User-Centered Information Visualization Process for Semantic Web Data“. In: *Proceedings of the 3rd International Workshop on Semantic Models for Adaptive Interacive Systems (SEMAIS)*.

Waltsgott, J. (Sep. 2009): „Clientseitige Integration von User Interface Services“. Diplomarbeit. Lehrstuhl für Multimediatechnik. Technische Universität Dresden.

Want, R., Hopper, A., Falcão, V. und Gibbons, J. (Jan. 1992): „The Active Badge Location System“. In: *ACM Transactions on Information Systems (TOIS)* 10.1, S. 91–102.

Wende, R. (Mai 2011): „Flexibilisierung eines Kommunikationsmodells für komponentenbasierte Webanwendungen“. Großer Beleg. Lehrstuhl für Multimediatechnik. Technische Universität Dresden.

Wiecha, C., Akolkar, R., Hosn, R. und Ling, T. (2006): „Mash-ups Considered Harmful! Composition and Choreography of Web Components“. In: *UIST '06 - Adjunct Proceedings of the 19th Annual ACM Symposium on User Interface Software and Technology*. ACM.

Winkler, R. (Sep. 2007): „Entwicklung eines ontologiebasierten Kontextmodells für kollaborative Web-Anwendungen“. Diplomarbeit. Lehrstuhl für Multimediatechnik. Technische Universität Dresden.

Wright, J. und Dietrich, J. (Jan. 2008): „Survey of existing languages to model interactive web applications“. In: *Proceedings of the 5th Asia-Pacific Conference on Conceptual Modelling*. Bd. 79. CRPIT. Australian Computer Society, Inc., S. 113–123.

Yao, H. und Etzkorn, L. (2004): „Towards a Semantic-Based Approach for Software Reusable Component Classification and Retrieval“. In: *Proceedings of the 42nd Annual Southeast Regional Conference*. ACM-SE 42. Huntsville, Alabama: ACM, S. 110–115. ISBN: 1-58113-870-9.

Young, O., Daley, E., Gualtieri, M., Lo, H. und Ashour, M. (Mai 2008): *The Mashup Opportunity*. Forrester.

Yu, J., Benatallah, B., Casati, F. und Daniel, F. (Sep. 2008): „Understanding Mashup Development“. In: *IEEE Internet Computing* 12.5, S. 44–52.

Yu, J., Benatallah, B., Saint-Paul, R., Casati, F., Daniel, F. und Matera, M. (2007): „A Framework for Rapid Integration of Presentation Components“. In: *WWW '07: Proceedings of the 16th International Conference on World Wide Web*. Banff, Alberta, Canada, S. 923–932. ISBN: 978-1-59593-654-7.

Zhang, Q., Cheng, L. und Boutaba, R. (2010): „Cloud computing: state-of-the-art and research challenges“. In: *Journal of Internet Services and Applications* 1.1, S. 7–18.